

王学颖 刘立群 刘冰 司雨昌 编著

Python学习
——从入门到实践

清华大学出版社

中国高校创意创新创业教育系列丛书

Python学习——从入门到实践

王学颖　刘立群　刘　冰　司雨昌　编著

清 华 大 学 出 版 社
北　京

内 容 简 介

本书是一本适合 Python 初学者学习程序设计与开发的基础教程，从应用的角度介绍了 Python 的发展、基本语句与语法、数据与运算、程序基本结构、函数与模块、面向对象和文件处理。本书既注重知识的系统性，又兼顾了内容的实用性，既保持了结构的严谨完整，又体现了语言的清晰简洁。

本书设置了丰富的教学案例，帮助读者用最简单直观的方式理解知识。同时，本书选取了 Python 常用的第三方库函数的应用实例，内容涉及图形绘制、中文分词、图形用户界面、网络爬虫、数据库访问等，引导读者进行深入的学习和研究。

本书内容具有知识完整、通俗易懂、叙述简练的特点，适合各层次读者使用，既可以作为高校计算机课程的教材，也可以供初学者或专业人士阅读。本书配套的电子资源包括 PPT、案例代码、习题等，均提供免费下载。

图书在版编目(CIP)数据

Python 学习：从入门到实践/王学颖等编著. —北京：清华大学出版社，2017
(中国高校创意创新创业教育系列丛书)
ISBN 978-7-302-48697-8

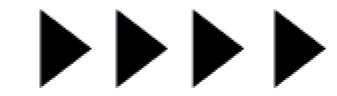
Ⅰ. ①P…　Ⅱ. ①王…　Ⅲ. ①软件工具－程序设计　Ⅳ. ①TP311.56

中国版本图书馆 CIP 数据核字(2017)第 270409 号

责任编辑：谢　琛
封面设计：常雪影
责任校对：焦丽丽
责任印制：李红英

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课 件 下 载：http://www.tup.com.cn，010-62795954
印 装 者：三河市君旺印务有限公司
经　　销：全国新华书店
开　　本：210mm×235mm　　**印　　张**：17.75　　**字　　数**：367 千字
版　　次：2017 年 12 月第 1 版　　**印　　次**：2017 年 12 月第 1 次印刷
印　　数：1～2000
定　　价：48.00 元

产品编号：075598-01



前　言

Python 语言是一种面向对象的解释型计算机程序设计语言，它既支持面向过程的编程，也支持面向对象的编程。Python 的语法简洁，没有过多的语法细节要求，其代码可读性强且更高效。Python 具有优秀的可拓展性，至今已有 11 万余个标准库和第三方库，可以方便地实现顶层和底层的黏性扩展，被称为胶水语言。Python 语言是一种完全开源的语言，因此被广泛使用，据 TIOBE 编程语言排行榜统计，截至 2017 年 5 月，Python 语言位于编程语言排行榜第四，仅次于 Java、C、C++ 语言。

“高级语言程序设计基础”是高校普遍开设的一门计算机基础课程，它面向计算机专业和非计算机专业的学生，主要目标是通过程序设计语言的学习，使学生掌握程序设计的基本思想和方法，培养和训练分析解决问题的思维习惯。Python 语言以其优美、清晰、简单的语法特点，非常适合作为第一门程序设计语言，它不仅非常容易掌握，更重要的是，Python 语言利用其丰富的函数库可以方便地开发面向各学科领域的应用，是学生进行专业学习和研究的有力工具。可以说，Python 是一种“一学就会”并使人终身受益的程序设计语言。

本书就是在上述背景下编写的，读者对象主要是编程零基础的学生。书中内容强调通俗易懂、简洁清晰、由浅入深。全书共分为 10 章，主要内容包括 Python 语言概述、数据类型和表达式、控制语句、数据结构、字符串和正则表达式、函数和模块、类和对象、文件处理、异常处理以及高级应用。

本书内容覆盖了 Python 的全部知识点，并且对每一个重要知识都设置了程序设计实例，强化对核心知识点的解读，引导学生通过具体案例掌握程序设计的方法。在案例的选择上，本书注重趣味性和实用性，使实例贴近生活、面向专业，既改变了程序设计的刻板生硬，又具有一定的实际应用价值。

本书由王学颖、刘立群、刘冰、司雨昌共同编著，在编写过程中参考了许多任课教师的意见和建议，在此向这些老师表示衷心的感谢。

本书提供了丰富的教学资源，内容包括教学 PPT、教学案例、习题和答案。

本书在写作过程中参考了大量的书籍和资料,在此向这些文献的作者表示衷心的感谢。

由于作者水平有限,书中难免有不足之处,敬请广大读者提出宝贵意见。

作　者

2017 年 5 月

目　录

第 1 章

Python 语言概述

学习目标

- 了解计算机语言的演变
- 了解高级语言的运行机制
- 掌握 Python 语言环境的安装与运行
- 掌握 Python 语言的基本语法



1.1 从计算机到编程

1.1.1 程序语言的演变

从第一台计算机宣告问世以来，程序就成为计算机的代名词。正如计算机所经历的 70 年飞速发展一样，编程也已经远远不是当初的样子。

编程其实就是把人类的需求用计算机语言表达，是人与计算机的对话。计算机语言 (Computer Language)是人与计算机之间通信的语言，是人与计算机之间传递信息的媒介。计算机语言经历了从机器语言、汇编语言到高级语言的演变过程。

机器语言是用二进制代码表示的计算机能直接识别和执行的一种指令的集合，它是计算机的设计者通过计算机的硬件结构赋予计算机的操作功能。机器语言是计算机唯一能够识别的语句，由 0 和 1 构成指令的集合。这样的机器语言十分复杂、不易阅读和修改，而且容易产生错误。程序员们很快就发现了使用机器语言带来的麻烦，它们难以辨别和记忆，给整个产业的发展带来了障碍，于是，汇编语言产生了，如图 1.1 所示。

```
1   操作：寄存器 BX 的内容送到 AX 中
2   1000100111011000        机器指令
3   MOV   AX, BX            汇编指令
```

图 1.1　从机器指令到汇编指令

汇编语言是经过符号化的计算机语言，它用一些简洁的英文字母和符号替代二进制指令，相对于枯燥的机器代码而言更易于读写、调试和修改。汇编语言需要一个专门的程序将这些符号翻译成相应的二进制机器指令，这种翻译程序称为汇编程序。汇编语言直接面向处理器，它与硬件关系密切，因此通用性和可移植性较差。也正因为如此，它的程序执行代码短、执行速度快、效率较高，因此在一些时效性要求较高的程序以及许多大型程序的核心模块和工业控制方面得到大量应用。

高级语言主要是相对于汇编语言而言的，是一种比较接近自然语言和数学公式的编程语言。高级语言用人们更易理解的方式编写程序，基本脱离了机器的硬件系统，有更强的表达能力，可方便地表示数据的运算和程序的控制结构，能更好地描述各种算法，而且易于学习和掌握。计算机语言的演变见表 1.1。

表 1.1 计算机语言的演变

计算机语言	编写方式及要素	特　点
机器语言	二进制编码 操作码、地址码	速度快，效率高，占用内存少 直观性差，难以纠错，编写需要很强的专业性
汇编语言	助记符号 操作码、地址码	速度快，效率高，占用内存少，直观性较强 编写专业性较强
高级语言	接近自然语言的语法 源程序、编译或解释程序	占用内存多，执行需要编译 易于掌握，可读性强 独立性、共享性及通用性强

例如，在高级语言中表示“将 BX 的内容送到 AX”，使用的代码为 AX=BX。

高级语言编写的程序称为高级语言源程序，但是高级语言并不是特指的某一种具体的语言，而是包括很多种编程语言，如流行的 Java、C、C++、C#、Pascal、Python 语言等，这些语言的语法以及命令格式都不尽相同。

1.1.2 高级语言的运行机制

高级语言按照执行方式可以分为编译型和解释型两种。

1. 编译型语言

编译型高级语言是通过专门的编译器，将高级语言代码(源程序)一次性翻译成可执行的机器码(目标程序)的编程语言，这个翻译过程称为 Compile。编译生成的可执行程序可以脱离开发环境，在特定的平台上独立运行。常用的编译型语言有 C、C++、FORTRAN 等。

编译程序对源程序进行解释的方法相当于日常生活中的整文翻译。在编译程序的执行过程中,要对源程序扫描一遍或几遍,最终形成一个可在具体计算机上执行的目标程序。通过编译后可以产生高效运行的目标程序,目标程序可以不依赖于程序环境而被多次执行。

编译型语言具有如下优点。

(1) 可独立运行,源代码经过编译形成的目标程序可脱离开发环境独立运行。

(2) 运行效率高,编译过程包含程序的优化过程,编译的机器码运行效率较高。

2. 解释型语言

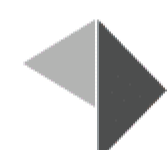
解释型语言是通过解释器对高级语言代码(源程序)逐行翻译成机器码并执行的语言。每次执行程序都要进行一次翻译,因此解释型语言的程序运行效率较低,不能脱离解释器独立运行。常用的解释型语言有 Basic、Python 等。

解释程序对源程序进行翻译的方法相当于日常生活中的同声传译。解释程序对源程序的语句从头到尾逐句扫描、逐句翻译,并且翻译一句执行一句,因而这种翻译方式并不形成机器语言形式的目标程序。

解释型语言的优点如下。

(1) 易于修改和测试,逐句解释过程中便于对代码进行修改和测试。

(2) 可移植性较好,只要有解释环境,就可在不同的操作系统上运行。



1.2 Python 的产生与特性

1.2.1 Python 语言的发展

Python 语言的作者 Guido von Rossum 是荷兰人。1982 年,Guido 在阿姆斯特丹大学(University of Amsterdam)获得了数学和计算机硕士学位。在那个时候,他接触并使用过诸如 Pascal、C、FORTRAN 等语言,这些语言的基本设计原则是让机器能更快地运行。在 20 世纪 80 年代,个人计算机的配置很低,如早期的 Macintosh 只有 8MHz 的 CPU 主频和 128KB 的 RAM,一个大的数组就能占满内存,所有编译器的核心是进行优化,以便让程序能够运行。为了提高效率,程序员需要像计算机一样思考,以便能写出更符合机器口味的程序。这种编程方式让 Guido 感到苦恼,他知道如何用 C 语言写出一个功能,但整个编写过程需要耗费大量的时间。Guido 受到 UNIX 系统的解释器 Bourne Shell 的启发,Python 应运而生。

Python 这个名字来自 Guido 所挚爱的电视剧 *Monty Python's Flying Circus*,他希望这个叫做 Python 的新语言,能符合他的理想:创造一种 C 和 Shell 之间的功能全面、易

学易用、可拓展的语言。

1991 年，第一个 Python 编译器(同时也是解释器)诞生了。它是用 C 语言实现的，并能够调用 C 库，已经具有了类(class)、函数(function)、异常处理(exception)，包括了列表(list)和词典(dictionary)等数据类型，以及以模块(module)为基础的拓展系统。

由于 Python 的开源性，使它经历了一个快速发展的阶段。

```
Python 2.0-2001/06/22
Python 2.4-2004/11/30
Python 2.5-2006/09/19
Python 2.6-2008/10/01
Python 2.7-2010/07/03
Python 3.0-2008/12/03
Python 3.1-2009/06/27
Python 3.2-2011/02/20
Python 3.3-2012/09/29
Python 3.4-2014/03/16
Python 3.5-2015/09/13
Python 3.6-2016/12/23
```

Python 3.0 相对早期的版本是一个较大的升级，但是 Python 3 在设计时没有考虑向下兼容，所以很多早期版本的 Python 程序无法在 Python 3 上运行。为了照顾早期的版本，Python 推出了过渡版本 2.6，它基本使用了 Python 2.x 的语法和库，同时考虑了向 Python 3.0 的迁移，允许使用部分 Python 3.0 的语法与函数。2010 年继续推出了兼容版本 2.7。2014 年 11 月，Python 发布消息，将在 2020 年停止对 2.0+版本的支持，并且不会再发布 2.8 版本，建议用户尽可能地迁移到 3.4+版本。

1.2.2 Python 语言的特性

Python 崇尚优美、清晰、简单，是一个优秀并广泛使用的语言。

1. 语法简单

Python 语法很多来自 C 语言，但又与 C 语言有很大的不同。Python 程序没有太多的语法细节和规则要求，采用强制缩进的方式。这些语法规定让代码的可读性更好，编写的代码质量更高，程序员能够更简单、高效地解决问题，使编程能够专注于解决问题而不是语言本身。

2. 可移植性

用 Python 编写的代码可以移植在许多平台上，这些平台包括 Linux、Windows、FreeBSD、Macintosh、Solaris、OS/2、Amiga、AROS、AS/400、BeOS、OS/390、z/OS、Palm OS、QNX、VMS、Psion、Acom RISC OS、VxWorks、PlayStation、Sharp Zaurus、Windows CE，甚至还有 PocketPC、Symbian 以及 Google 基于 Linux 开发的 Android 平台等。

3. 黏性扩展

Python 又被称为胶水语言，它具有优秀的可拓展性。Python 可以在多个层次上拓展，既可以在高层引入 py 文件，也可以在底层引用 C 语言的库。Python 已经拥有 11 万余个标准库和第三方库，它可以完成的操作包括正则表达式、文档生成、单元测试、线程、数据库、网页浏览器、GUI(图形用户界面)等操作。因此，Python 的编程就像钢结构房屋一样，程序员可以在此框架下自由地任意搭建功能。

4. 开源理念

Python 语言是一种开源语言，使用者可以自由地发布这个软件的复制、阅读它的源代码、对它进行改动、把它的一部分用于新的自由软件中等。正是由于 Python 的完全开源，Python 吸引了越来越多优秀的人加入进来，形成了庞大的 Python 社区。如今，各种社区提供了成千上万的开源函数模块，而且还在不断地发展。

5. 面向对象

Python 既支持面向过程的函数编程，也支持面向对象的抽象编程。在面向过程的语言中，程序是由过程或可重用代码的函数构建起来的，而在面向对象的语言中，程序是由数据和功能组合而成的对象构建起来的。与其他主要的语言(如 C++ 和 Java)相比，Python 以一种非常强大而简单的方式实现面向对象编程。



1.3 Python 的安装与运行

1.3.1 Python 的下载和安装

搭建 Python 的开发环境，就是指安装 Python 的解释器。IDLE 是开发 Python 程序的基本 IDE(集成开发环境)，具备基本的 IDE 功能，是非商业 Python 开发的不错选择。只要安装 Python 以后，IDLE 就自动安装好了。

Python 的安装非常简单。首先,需要进入 Python 官方网站 http://www.python.org/downloads/下载适合操作系统的安装包,如图 1.2 所示。

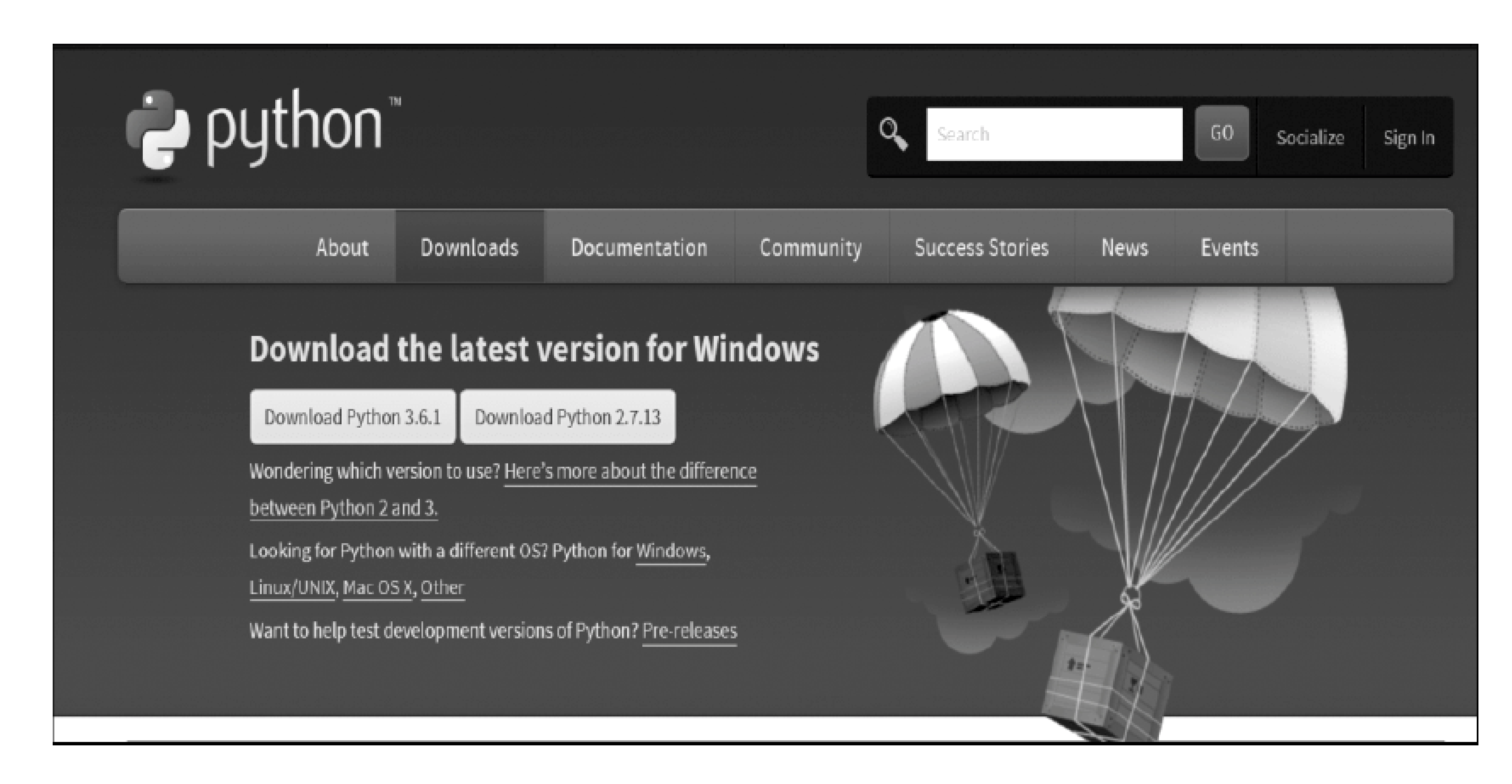


图 1.2　Python 官方网站

网页页面会显示发布的最新版本,会随着陆续的发布随时进行更新。在这里要根据相应的操作系统选择不同的版本,如选择 Python For Windows,如图 1.3 所示。在列出的版本列表中选择一个版本,单击即可下载。

Python releases by version number:

Release version	Release date	
Python 3.6.1	2017-03-21	Download
Python 3.4.6	2017-01-17	Download
Python 3.5.3	2017-01-17	Download
Python 3.6.0	2016-12-23	Download
Python 2.7.13	2016-12-17	Download
Python 3.4.5	2016-06-27	Download
Python 3.5.2	2016-06-27	Download

图 1.3　Python 官网发布的更新

Python 3 对 Python 2 进行了重大升级,Python 已经不再进行 Python 2 的后续更新,Python 3 不完全向下兼容 2.x 程序。本书所有程序和案例全部基于 Python 3,建议初学者选择 3.x 版本进行下载并安装,如图 1.4 所示。

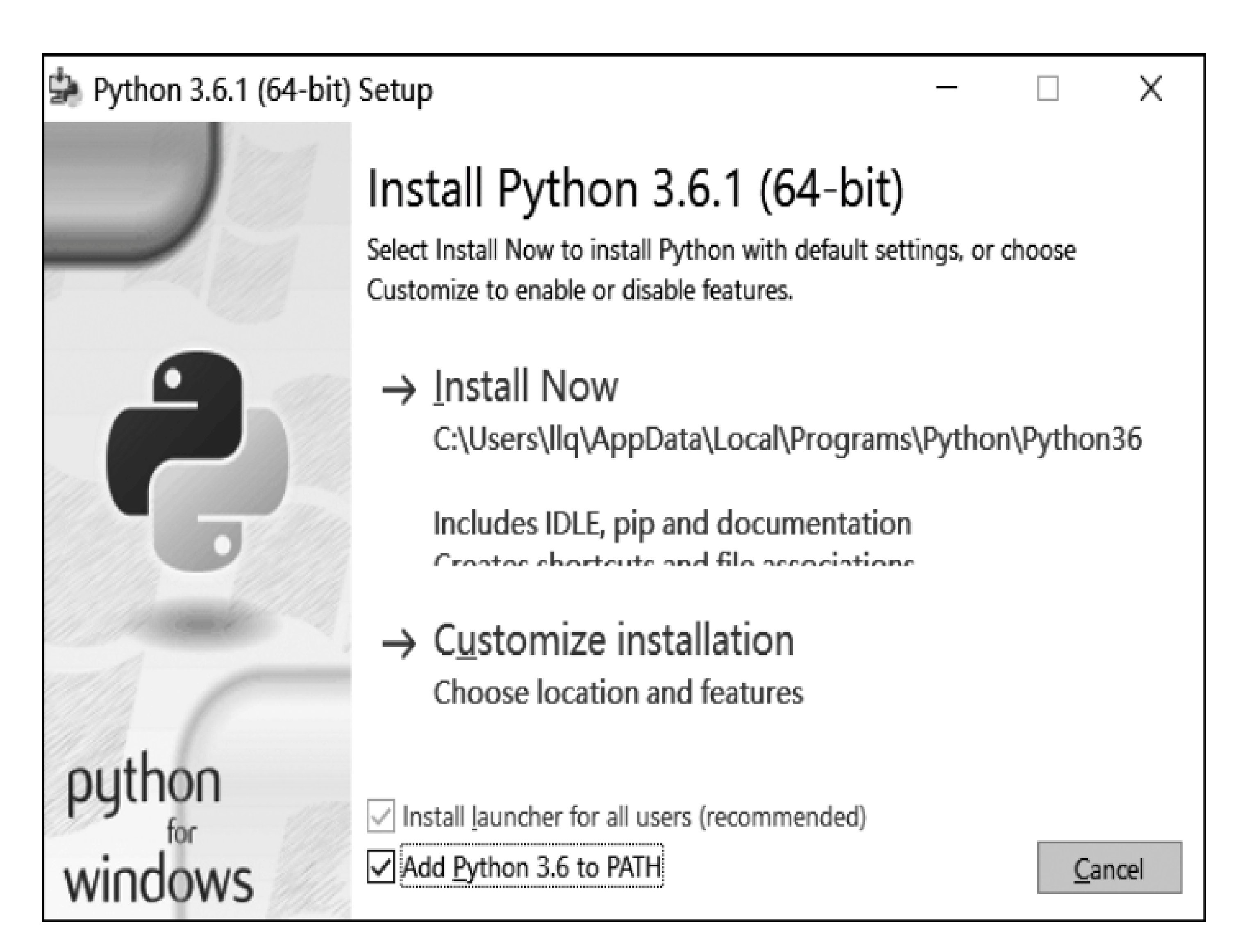


图 1.4　Python 安装

安装程序会在默认安装目录下安装 Python.exe、库文件及其他文件。这里请注意，为了能够在 Windows 命令行窗口直接调用 Python 文件，需要在安装时将 Python 安装目录添加到系统 Path。请在安装时选中复选框 Add Python 3.6 to PATH。安装过程非常简单，只要按照安装向导，单击"下一步"就可以了。安装成功界面如图 1.5 所示。

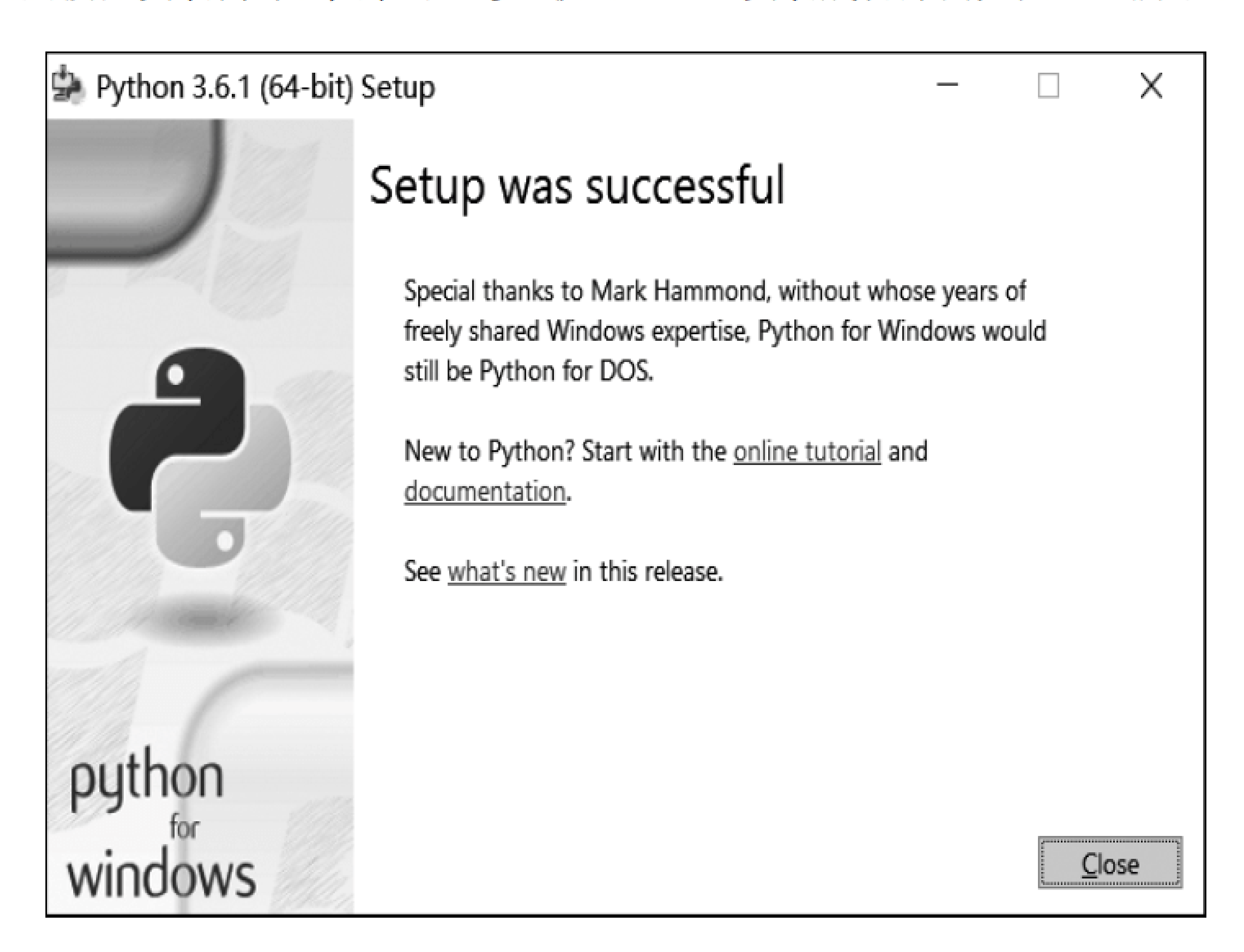


图 1.5　Python 安装成功

将 Python 安装包下载到本地并安装好之后，在开始菜单找到 IDLE (Python 3.6)并单击打开，就可以进行编程了。IDLE 是标准的 Python 内置的集成开发环境，由 Guido van Rossum 亲自编写。本书将首先采用 IDLE 作为开发环境，同时，Python 也支持其他的第三方开发环境。

以下是几款比较常用的第三方开发环境：

```
Pycharm
VIM
Eclipse with PyDev
Sublime Text
Komodo Edit
PyScripter
Interactive Editor for Python
```

1.3.2 Python 的运行

Python 安装完成后，在 Windows 的“开始”菜单中选择“程序”→Python 3.6→IDLE (Python 3.6 64bit)，即可启动内置的解释器(IDLE 集成开发环境)，如图 1.6 所示。

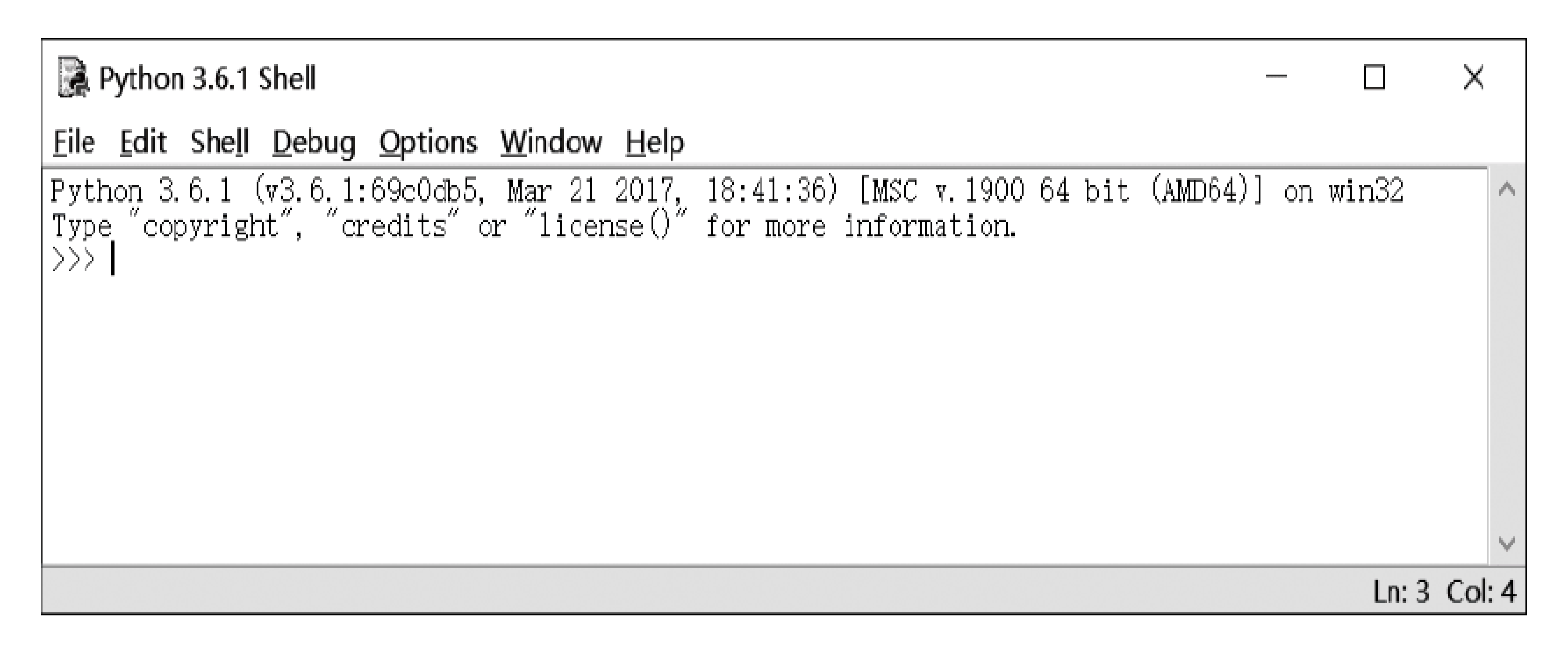


图 1.6 IDLE 集成开发环境

在 Python 解释器(IDLE)中可以采用命令行方式和文件执行方式两种运行方式。

1. 命令行方式

命令行方式是一种交互式的命令解释方式，当输入一条命令后，解释器(Shell)即负责解释并执行命令。例如，直接在提示符(>>>)后输入语句，下一行将显示出命令的输出结果，如图 1.7 所示。

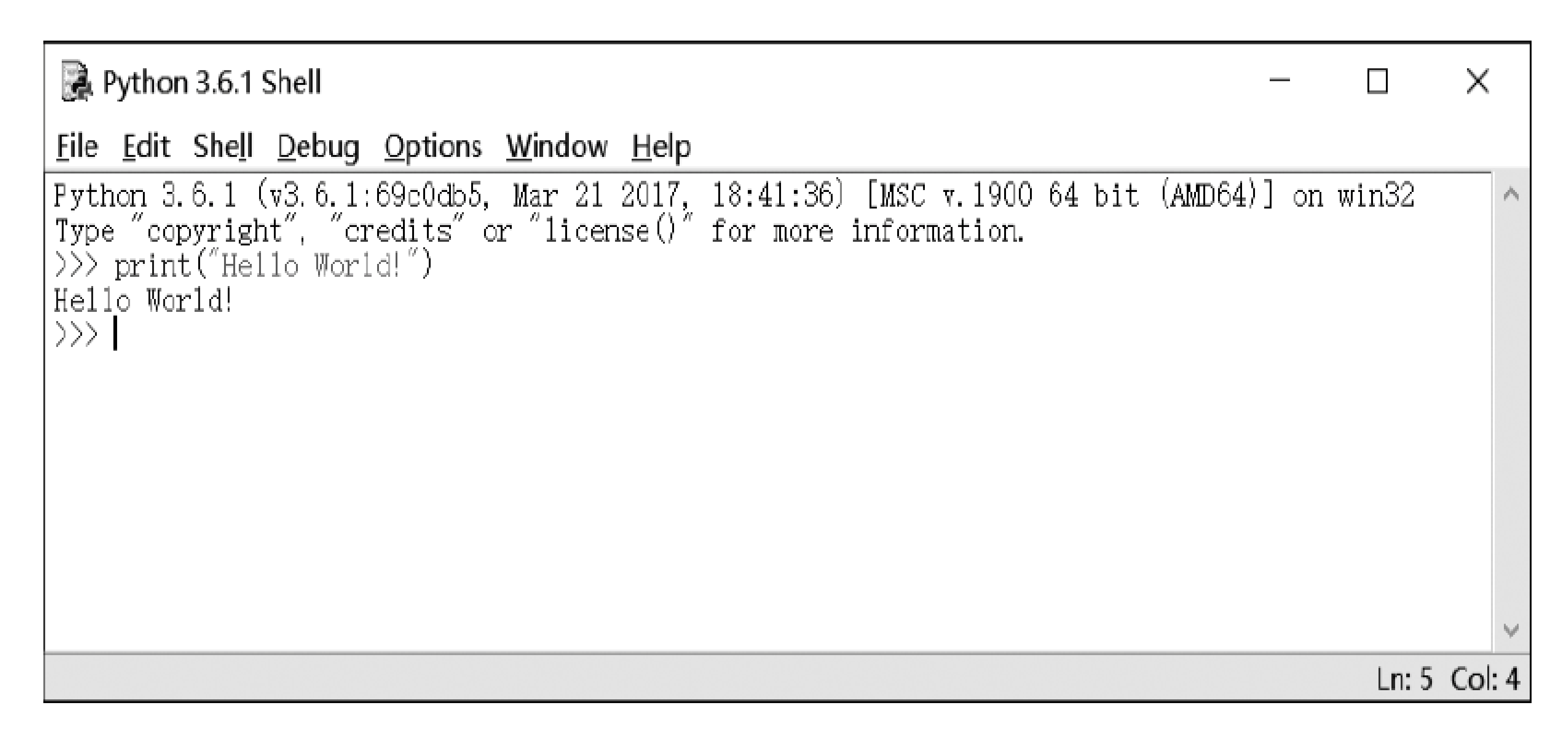


图 1.7　IDLE 命令行方式

下面语句的作用是打印输出一行文本“Hello World!”。

```
>>>print("Hello World!")
```

2. 文件执行方式

文件执行方式是在解释器中建立程序文件(以 py 为扩展名),然后调用并执行该文件。

(1) 如图 1.8 所示,在解释器的 File 菜单中选择 New File 新建一个文件,将命令写入并保存到文件 Hello.py。

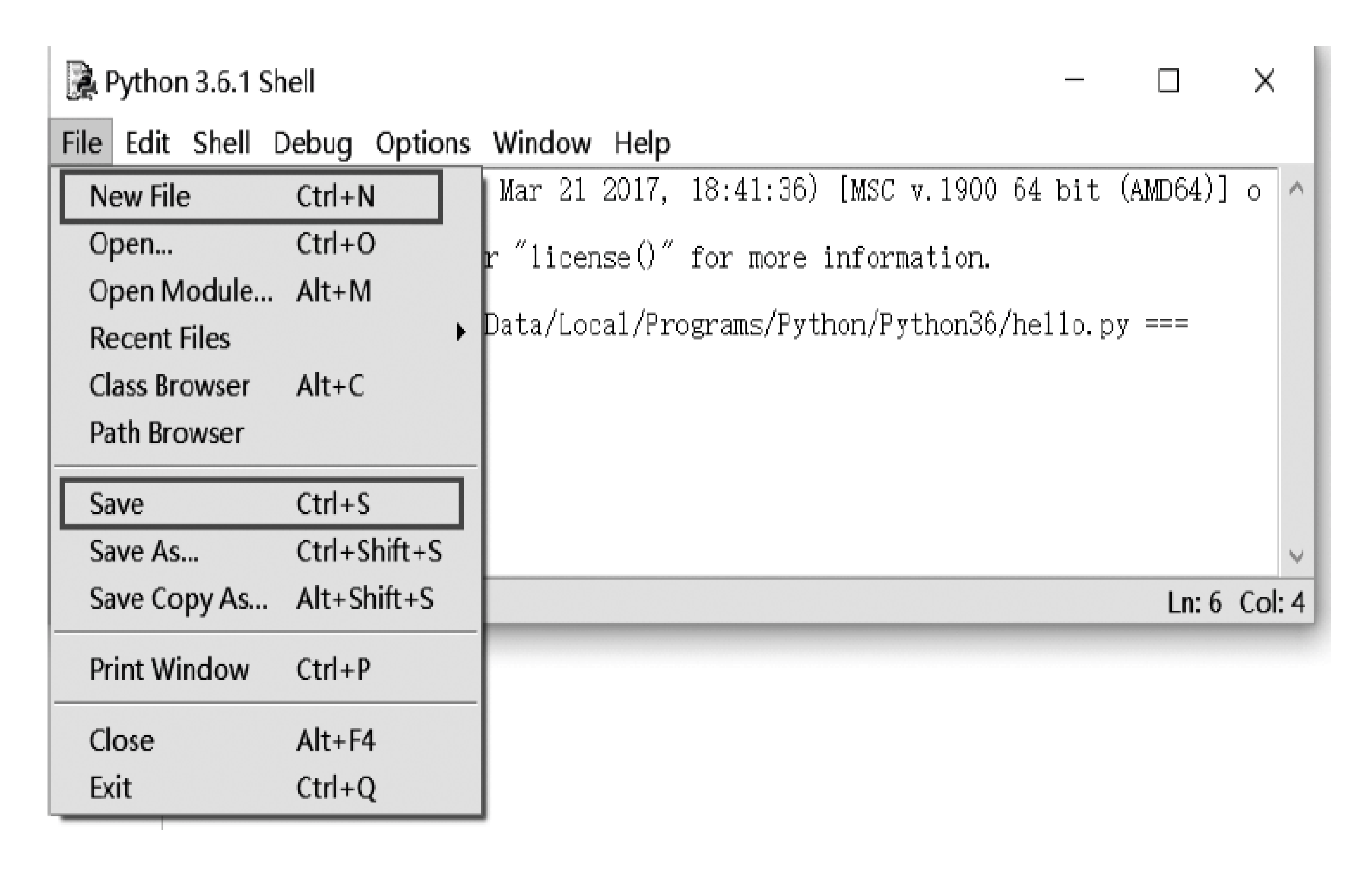


图 1.8　IDLE 文件执行方式

(2) 如图 1.9 所示,在文件 Hello.py 的窗口中,选择 Run→Run Module 命令,即可

在 Python 的解释器中运行程序。运行结果如图 1.10 所示。

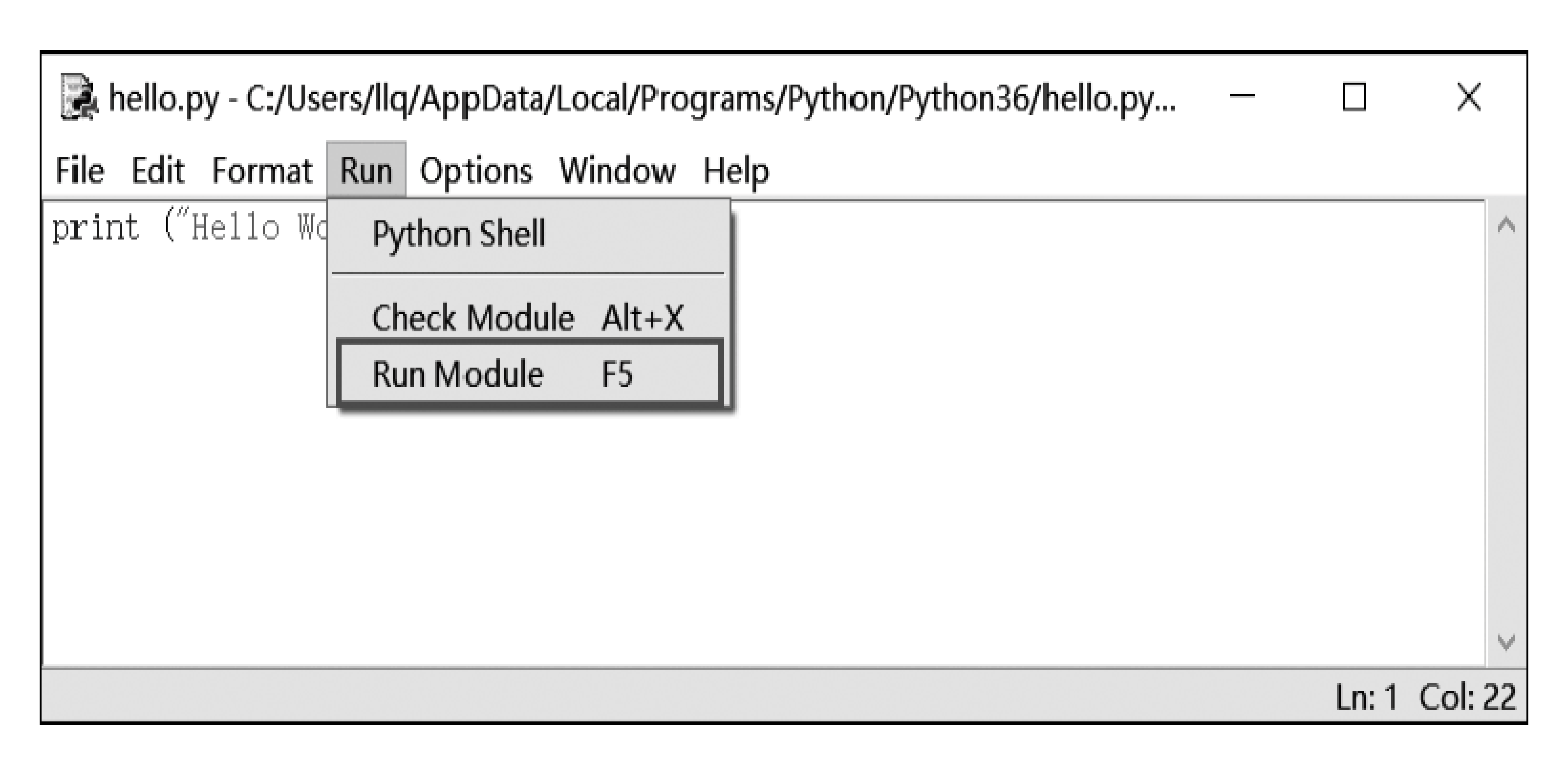


图 1.9 文件执行过程

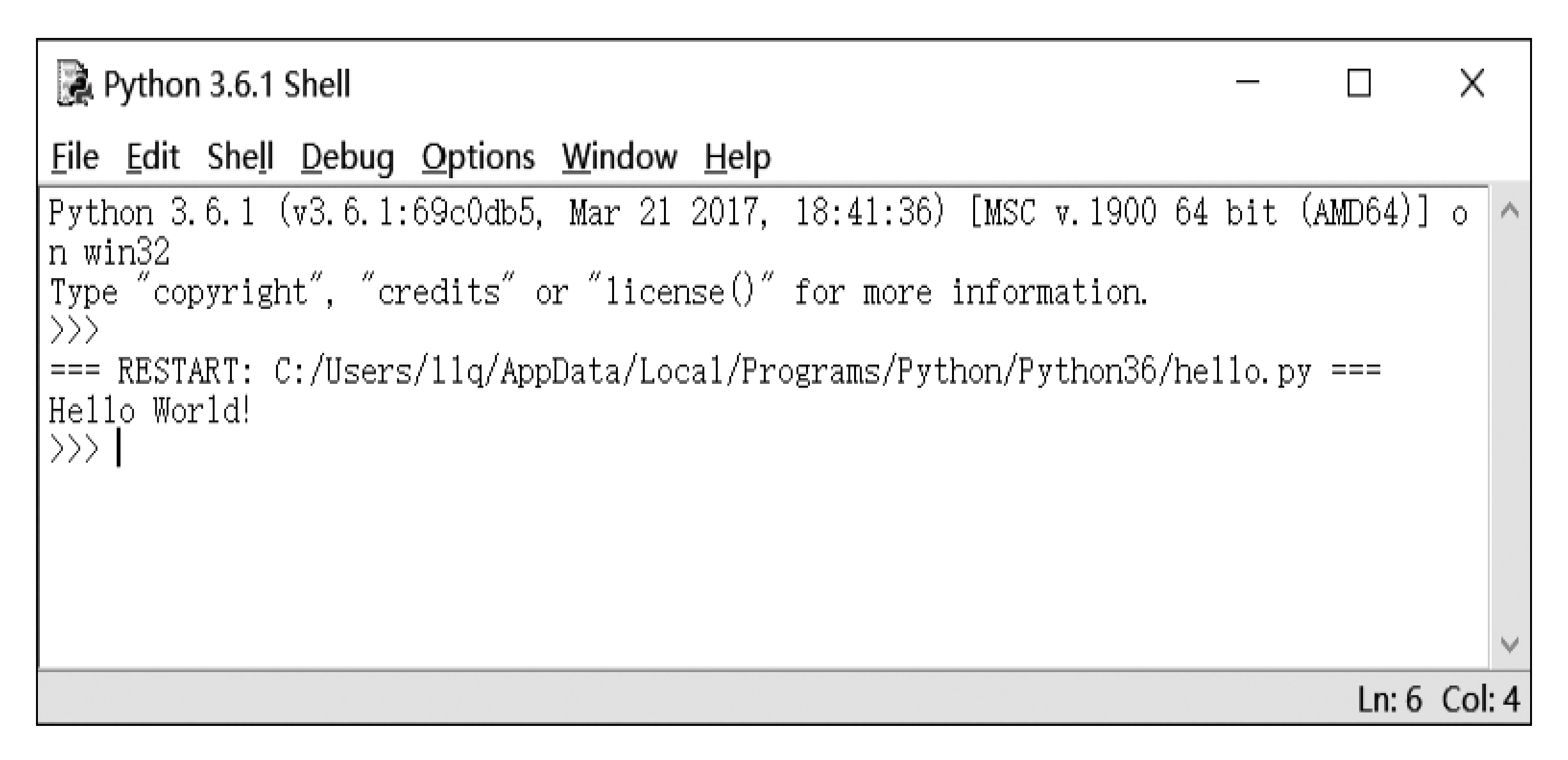


图 1.10 文件执行结果

(3) 若要打开已经存在的程序文件并运行,可在解释器(IDLE)中,选择 File 菜单→Open File 打开一个文件,在打开的文件窗口中选择 Run→Run Module 命令,就可以运行此程序。



1.4 Python 的基础语法

1.4.1 程序的基本结构

本节通过两个简单的实例程序,介绍 Python 程序的构成和基本语法规则。

【例1.1】 计算圆的面积。

```
#example1.1
pi=3.1415926                    #定义常量pi
r=eval(input("请输入圆的半径:"))   #输入圆的半径
s=pi*r*r                        #计算圆的面积
print("半径为",r,"圆的面积为:",s)   #输出计算结果
```

程序运行结果如下:

```
>>>
====RESTART: C:\Users\Python\example1.1.py====
请输入圆的半径:25
半径为 25 圆的面积为:1963.4953750000002
>>>
```

说明:

(1) 程序的功能是输入圆的半径,计算圆的面积。

(2) 代码中input()是输入函数,返回用户输入并赋值给变量r。

(3) eval()函数用来将输入的字符转换为数值。

(4) s=pi*r*r,用表达式计算圆的面积,这里的"="是赋值运算。

(5) print()是输出函数,用来显示计算结果。

【例1.2】 求一元二次方程的实根。

```
#example1.2
a=eval(input("请输入a:")) #输入a
b=eval(input("请输入b:")) #输入b
c=eval(input("请输入c:")) #输入c
d=b**2-4*a*c
if d>=0:                         #判断是否有实根
    x1=(-b+d**0.5)/(2*a)
    x2=(-b-d**0.5)/(2*a)
    print('方程的根:x1=%f,x2=%f'%(x1,x2))
else:
    print('input data Error!')
```

程序两次运行结果如下:

```
>>>
===========RESTART: C:\Users\Python\example1.2.py============
请输入 a: 1
请输入 b:-2
请输入 c: 1
方程的根: x1=1.000000,x2=1.000000
>>>
===========RESTART: C:\Users\Python\example1.2.py============
请输入 a: 1
请输入 b: 2
请输入 c: 3
input data Error!
>>>
```

说明：

(1) 程序用来求一元二次方程 $ax^2+bx+c=0$ 的实数根。

(2) a、b、c 为方程的三个系数，由 input()函数返回用户输入。

(3) d=b ** 2-4 * a * c，计算 $d=b^2-4ac$ 的值，此处 ** 是 Python 的乘方运算符。

(4) if…else 用来实现程序的分支，当 d≥0 时，计算方程的两个实根，否则显示信息提示错误。

(5) print(' 方程的根：x1=%f,x2=%f'%(x1,x2))语句中，使用了格式符控制显示数据的样式。

Python 程序的基本框架可以概括为 IPO 结构，即数据输入(input)、数据处理(process)、数据输出(output)。高级语言的程序都是用来求解特定问题的，而问题的求解都可以归结为计算问题，IPO 的程序结构正是反映了实际问题的计算过程。

数据输入是问题求解所需数据的获取，可以由函数、文件等完成输入，也可以由用户输入，如例 1.1 中的 input 语句用来返回用户输入的圆的半径。

数据处理是对输入数据进行计算。这里的运算可以是数值运算、文本处理、数据库等。可以使用运算符、表达式实现简单的运算，或者调用函数、方法完成复杂运算。如例 1.1 中计算圆的面积的语句 s=pi * r * r。对于复杂处理过程可以使用程序的分支和循环，如例 1.2 中的 if 语句。

数据输出是显示输出数据计算的结果，有图形输出、文件输出等输出形式。上面实例中 print 语句将结果输出到屏幕上。print 语句有标准格式输出和函数输出的不同形式，如例 1.2 中的语句，print('方程的根：x1=%f,x2=%f'%(x1,x2))，用来在屏幕上显示程序的运算结果。

1.4.2 基本语法规则

1. 注释

注释用来在程序中对语句、运算等进行说明和备注。适当地添加注释语句可以很好地增加程序的可读性,同时也便于代码的调试和纠错。

Python 中用#符号表示单行注释,用'''(3 个单引号)表示多行注释。注释语句仅是说明性文字,不会作为代码而被执行。在 IDLE 窗口中,注释语句以红色或绿色文字标出,用以区别代码部分。

2. 标识符

标识符用来表示常量、变量、函数、对象等程序要素的名字。例 1.1 中变量 r 用来存放圆的半径,变量 s 用来存放圆的面积,它们都是表示变量名字的标识符。

给变量命名时,可以使用任何能够明确说明该数据特征、用途、性质的标识符,可以包括由字母、数字、下画线或汉字组成的字符串。标识符要符合下面的命名规则。

(1) 首字符必须是字母、汉字或下画线。

(2) 中间可以是字母、下画线、数字或汉字,不能有空格。

(3) 区分大小写字母(如大写 S 和小写 s 代表了不同的两个变量)。

(4) 不能使用 Python 的关键字。

3. 关键字

关键字(Keyword)又称保留字,是 Python 系统内部定义和使用的特定标识符。每种程序设计语言都有自己的关键字。Python 3 中的关键字如下:

```
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del
', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is
', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield
']
```

4. 强制缩进

Python 最具特色的就是使用缩进表示代码块,不需要使用大括号{}。缩进的空格数是可变的,但是同一个代码块的语句必须包含相同的缩进空格数。

如例 1.2 中的 if 分支语句:

```
if d>=0:                    #判断是否有实根
    x1=(-b+d**0.5)/(2*a)
    x2=(-b-d**0.5)/(2*a)
    print('方程的根:x1=%f,x2=%f'%(x1,x2))
else:
    print('input data Error!')
```

以下代码由于最后一行语句缩进的空格数不一致,将会导致程序的运行错误:

```
if True:
    print ("Answer")
    print ("True")
else:
    print ("Answer")
  print ("False")           #缩进不一致,会导致运行错误
```

以上程序由于缩进不一致,执行后会出现类似下面的错误:

```
File "test.py", line 6
    print ("False")
IndentationError: unindent does not match any outer indentation level
```

5. 多行语句

Python 通常是一行写完一条语句,但如果语句过长,可以使用反斜杠“\”实现多行语句。例如:

```
total=item_one+\
item_two+\
item_three
```

在[],{}或()中的多行语句,不需要使用反斜杠“\”,例如:

```
total=['item_one', 'item_two', 'item_three',
        'item_four', 'item_five']
```

6. 同一行显示多条语句

Python 可以在同一行中使用多条语句,语句之间使用分号“;”分割。例如:

```
>>>import sys; x='goodday'; sys.stdout.write(x+'\n')
```

相当于分别执行了三行命令,代码输出结果如下:

```
goodday
8
```

7. import 与 from…import

在 Python 中,模块是一个包含所有定义的函数和变量的文件,其后缀名是 py。模块可以被其他程序引入,实现调用该模块中函数的功能,用 import 或者 from…import 导入。import 语句既可以导入一个自定义模块,也可以导入 Python 标准模块。

(1) 将整个模块导入的格式

```
import <模块名 1>[,<模块名 2>[,…<模块名 N>]
```

(2) 从某个模块中导入某个函数的格式

```
from <模块名> import <函数名>
```

(3) 从某个模块中导入多个函数的格式

```
from <模块名> import<函数名 1>[,<函数名 2>[,…<函数名 N>]
```

(4) 将某个模块中的全部函数导入的格式

```
from <模块名> import *
```

【例 1.3】 导入 Python 标准库模块。

```
#example1.3
#import 导入标准模块 sys
import sys
print('命令行参数如下:')
print(sys.argv)
for i in sys.argv:
    print(i)
```

程序运行结果如下：

```
>>>
============RESTART: C:/Users/Python /example1.3.py============
命令行参数如下：
['C:/Users/ Python /example1-3.py']
C:/Users/ Python /example1-3.py
```

说明：

(1) import sys 导入 Python 标准库的 sys.py 模块。

(2) 参数 sys.argv 用来返回一个包含命令行参数的列表。

习题 1

一、填空题

1. Python 程序文件的扩展名为________。
2. Python 有两种运行方式：________方式和________方式。
3. 高级语言的程序基本结构 IPO，分别指________、________、________。
4. Python 中使用________表示代码块，不需要使用大括号。
5. Python 中使用________注释语句和运算。
6. Python 中的 import 语句的作用是________。

二、判断题

1. Python 的标识符首字符可以是数字、字母或下画线。 (　　)
2. Python 的标识符区分字母的大小写，大写 X 和小写 x 代表不同的变量。 (　　)
3. Python 3.x 完全兼容 Python 2.x。 (　　)
4. 为了让代码更加紧凑，编写 Python 程序时应尽量避免加入空格和空行。 (　　)

三、简答题

1. 简述 Python 语言的特性。
2. 简述程序的 IPO 结构。

第 2 章

Python 数据类型和表达式

学习目标

- 了解 Python 的基本数据类型
- 掌握变量的概念和赋值方法
- 熟悉 Python 的运算符和表达式
- 掌握常用内置函数的使用方法
- 掌握常用标准函数库的导入和使用



2.1 基本数据类型

Python 中的数据类型包含基本数据类型和复合数据类型两种，其中基本数据类型有数值类型、字符串型和布尔型；复合数据类型有元组、列表和字典。本节仅介绍基本数据类型及其运算，复合数据类型将在后续章节介绍。

2.1.1 数值类型

数值型数据用于存储数值，用来参与算术运算。Python 支持的数值型数据有整型、浮点型和复数型。

1. 整型

整型是带正负号的整数数据。Python 3.x 中并不严格区分整型和长整型，且没有长度限制，表示范围仅与计算机支持的内存大小有关，所以它几乎包括了全部整数范围，远远超过了其他高级语言中整型数据的表示范围，给数据计算带来了很大的便利。

Python 整型数的表示方法有以下几种。

(1) 十进制整数，如 10,255,-16。

(2) 二进制整数,以 0B 或 0b 开头的数据,如 0B11,0b100。
(3) 八进制整数,以 0O 或 0o 开头的数据,如 0O67,0o17。
(4) 十六进制整数,以 0X 或 0x 开头的数据,如 0X4a,0xff。

2. 浮点型

浮点数表示实数数据,由整数部分、小数点和小数部分组成。使用下面的语句可以输出当前系统下浮点数所能表示的最大数 max 和最小数 min。

```
>>>import sys
>>>sys.float_info.max
1.7976931348623157e+308
>>>sys.float_info.min
2.2250738585072014e-308
>>>
```

Python 中的浮点数的表示方法如下。

(1) 十进制小数表示法,如 3.14,10.0,0.0 等,注意,这里的 0.0 不是 0,0 表示一个整数,而 0.0 表示一个浮点数。

(2) 科学计数表示法,用字母 e(或 E)表示以 10 为底数的指数,用 XeY 表示 $X*10^Y$。例如:

```
>>>1.23456e10
12345600000.0
>>>123.456e-2
1.23456
>>>
```

3. 复数型

复数型数据用来表示数学中的复数,复数由实数部分和虚数部分所组成,形如 x=a+bj。其中 a、b 为浮点数,j 为虚数单位,j 的平方等于-1。a 是复数的实部,b 是复数的虚部。

可以使用 x.real 和 x.imag 获得复数 x 的实部和虚部。例如:

```
>>>x=12.3+45j
>>>x
(12.3+45j)
```

```
>>>x.real
12.3
>>>x.imag
45.0
>>>
```

2.1.2　字符串类型

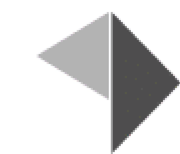
Python 语言中的字符串类型是用引号括起来的一个或多个字符。用单引号(')和双引号(")括起来的字符串必须是单行字符串,用三引号(''')括起来的可以是多行字符串。例如:

```
>>>str='God Wants To Check The Air Quality'
>>>str1="God Wants To Check The Air Quality"
>>>str2='''God
Wants To Check
The Air Quality'''
>>>
```

2.1.3　布尔类型

布尔型数据用来表示具有两个确定状态的数据,它有真(True)和假(False)两个值。布尔型数据在计算机中用 1、0 存储,1 代表逻辑真(True),0 代表逻辑假(False)。Python 中任何值为 0 或空的数据,如一个空字符串、一个空的元组等,它们的布尔值均为 False。

```
>>>x=True
>>>int(x)
1
>>>y=False
>>>int(y)
0
>>>
```



2.2 常量与变量

2.2.1 常量

在程序运行过程中，其值不发生改变的数据对象称为常量。常量是指一旦初始化后就不能修改的固定值，按其值的类型分为整型常量、浮点型常量、字符串常量等。例如：0，-255，3.1415926，"Python"，True。

使用 type()函数可以查看数据的类型。例如：

```
>>>type("Python")
<class 'str'>
>>>type(True)
<class 'bool'>
>>>type(3.1415926)
<class 'float'>
>>>type(0)
<class 'int'>
>>>
```

在 C 和 C++ 等高级语言中通常使用 const 关键字定义常量，而在 Python 中并没有类似的定义常量的关键字，但是 Python 可以通过定义对象类的方法创建。

2.2.2 变量

在程序运行过程中，可以随着程序的运行而更改的量称为变量。变量对应计算机内存中的一块区域，用来存储数据的值，不同类型的数据所分配的存储空间不同。Python 中的变量不需要声明，变量的赋值操作即是变量的声明和定义的过程。变量名由字母、汉字、数字或下画线组成，变量名的首字符不能是数字，且不能为 Python 的关键字。变量命名要符合标识符的命名规则，具体参见第 1.4.2 节。

1. 高级语言中的变量

计算机的内存是以字节为单位的存储区域，为了便于访问，每个存储单元有自己的编码，称为内存地址。通过内存地址访问存储空间，就如同人们通过门牌号码投递邮件一样。而实际上，程序要访问内存中的数据并不需要知道其物理地址，这是因为在高级语言

中,用变量名指向这个物理地址。

变量是指一个特定的存储空间,即一定字节数的内存单元。这一组存储单元用来存放指定的数据,而数据是可以随时变化的。通常情况下,在使用变量之前需要定义变量,定义变量就是定义变量的名称、类型和值。变量的类型决定了分配的内存单元的多少,即多少个字节。变量的名称就是对这一组内存单元的引用,这样,程序员就不必关心数据具体的存储地址,只要使用变量名就可以访问数据了。变量值发生改变时,改变的是存储单元中的内容,而变量的地址是不变的,如图 2.1 所示。

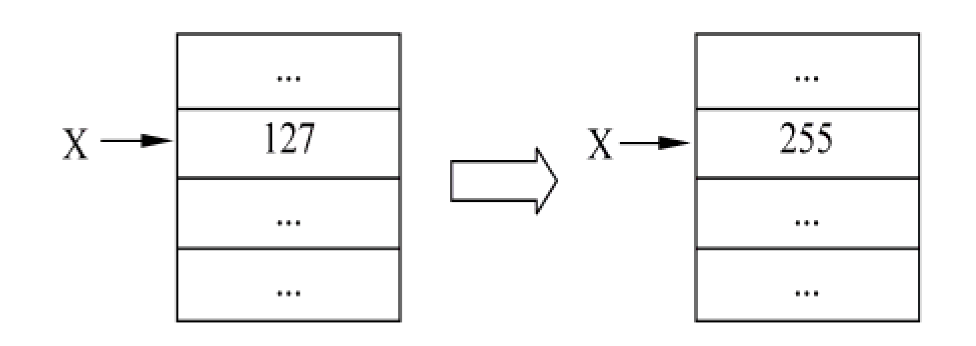


图 2.1　高级语言中的变量是地址的引用

2. Python 中的变量

Python 中的变量与其他高级语言中的变量不同。Python 语言没有专门定义变量的语句,而是使用赋值语句完成变量的定义。当用赋值语句给变量赋值的同时就定义了变量。变量名并不是对内存地址的引用,而是对数据的引用。也就是说,用赋值语句对变量重新赋值时,Python 为其分配了新的内存单元,变量将指向新的地址,变量的地址发生了变化,如图 2.2 所示。

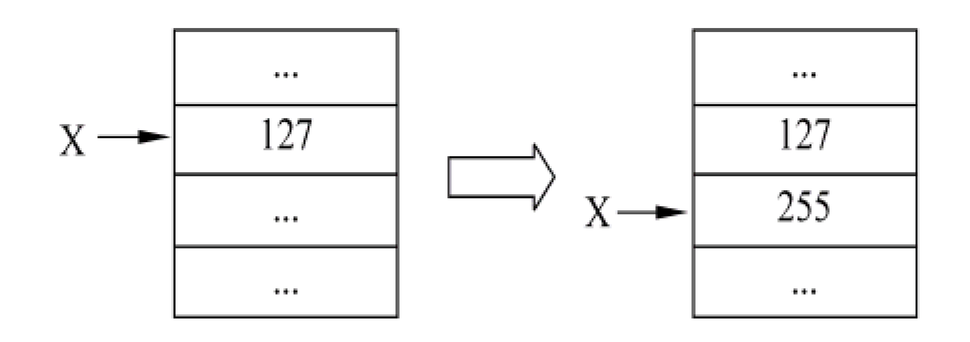


图 2.2　Python 中的变量是对数据的引用

使用 id 函数可以查看变量的内存地址。例如:

```
>>>x=127
>>>id(x)
1412875264
>>>x=255
>>>id(x)
```

```
1412879360
>>>
```

第 1 行代码定义了一个名为 x 的变量,该变量的初始值为 127。

第 2 行代码,输出变量 x 的地址。

第 3 行代码再次定义了一个名为 x 的变量,该变量的初始值为 255。

第 4 行代码,输出变量 x 的地址。

从上例中,可以发现变量 x 两次输出的内存地址并不相同,也就是说,Python 中的变量名是对数据的引用。

变量没有赋值就使用会出现错误,如下例中,在变量 z 没有赋值的前提下,不能直接输出 z 的值,否则会提示运行错误。

```
>>>print(z)
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in<module>
    print(z)
NameError: name 'z' is not defined
>>>
```

2.2.3 变量的赋值

Python 中的赋值语句用来定义变量的类型和数值。赋值语句有多种形式,包括一般形式、增量赋值、链式赋值以及多重赋值等。

1. 一般形式

在 Python 中,赋值号就是等号“=”。赋值语句的一般形式如下:

```
<变量名>=<表达式>
```

赋值号的左边必须是变量名,右边则是表达式。赋值语句先计算表达式的值,然后使该变量指向该数据。

```
>>>a=0                       #变量 a 指向数据 0
>>>b=1                       #变量 b 指向数据 1
>>>print(id(a),id(b))
1412871360 1412871456        #变量 a 变量 b 分别指向不同的数据
```

```
>>>a=b                          #变量 a 指向变量 b 数据
>>>print(id(a),id(b))
1412871456 1412871456           #变量 a 变量 b 分别指向相同的数据
>>>
```

2. 增量赋值

Python 中提供了 12 种增量赋值运算符：

```
+=、-=、*=、/=、//=、%=、**=、<<=、>>=、&=、|=、^=
```

例如：赋值语句 x+=5 相当于 x=x+5，如下例。

```
>>>x=10               #x 赋值为 5
>>>x+=5               #相当于 x=x+5
>>>x                  #x 的当前值为 15
15
>>>x*=5               #相当于 x=x*5
>>>x                  #x 的当前值为 75
75
>>>x/=5               #相当于 x=x/5
>>>x
15                    #x 的当前值为 15.0
```

3. 链式赋值

链式赋值的语句形式如下：

```
<变量 1>=<变量 2>=…=<变量 n>=<表达式>
```

链式赋值用于为多个变量赋一个相同的值。链式赋值先计算最后的表达式的值，然后将变量全部指向此数据对象。下例中的 x,y,z 指向同一个数据对象 3.1415926。

```
>>>x=y=z=3.1415926              #将 3.1415926 同时赋值给 x,y,x
>>>x
3.1415926
>>>y
3.1415926
```

```
>>>z
3.1415926
>>>
```

4. 多重赋值

多重赋值语句的形式如下：

```
<变量 1>,<变量 2>,…,<变量 n>=<表达式 1>,<表达式 2>,…<表达式 n>
```

赋值号两边的变量和表达式数量要一致，多重赋值首先计算赋值号右边的各表达式的值，然后按照顺序分别赋值给左边的变量。

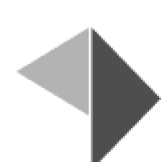
```
>>>a,b=3,5
>>>print(a)              #多重赋值的结果 a=3
3
>>>print(b)              #多重赋值的结果 b=5
5
>>>a,b=3,a               #多重赋值 a=3,b=a
>>>print(a)
3
>>>print(b)
3
>>>
```

使用多重赋值语句可以方便地实现两个变量的数据交换。例如：

```
>>>a,b=3,5
>>>a,b=b,a          #多重赋值实现 a,b 的数据交换
>>>print(a)
5
>>>print(b)
3
>>>
```

正是由于 Python 中变量名是对数据的引用，所以一条多重赋值语句就可以完成两个变量的数据交换，体现了 Python 语言简洁高效的风格。在其他高级语言中，若要实现两个变量 a 和 b 的数据交换，需要使用第三个变量 t，通过三条语句实现。例如：

```
>>>t=a
>>>a=b
>>>b=t
```



2.3　运算符与表达式

2.3.1　算术运算符

Python 中的算术运算符和运算规则如表 2.1 所示。

表 2.1　算术运算符和运算规则

运算符	描　　述	实例(a=10,b=20)
+	加,两个数据相加	a+b 输出结果 30
-	减,得到负数或是一个数减去另一个数	a-b 输出结果-10
*	乘,两个数相乘或是返回一个被重复若干次的字符串	a * b 输出结果 200
/	除,x 除以 y	b/a 输出结果 2
%	取模,返回除法的余数	b %a 输出结果 0
**	幂,返回 x 的 y 次幂	a * * b 为 10 的 20 次方 输出结果 100000000000000000000
//	整除,返回商的整数部分	9//2 输出结果 4 9.0//2.0 输出结果 4.0

2.3.2　关系运算符

Python 中的关系运算符和运算规则如表 2.2 所示。

表 2.2　关系运算符和运算规则

运算符	描　　述	实例(a=10,b=20)
==	等于,比较对象是否相等	(a==b)返回 False
!=	不等于,比较两个对象是否不相等	(a!=b) 返回 True
>	大于,返回 x 是否大于 y	(a>b)返回 False
<	小于,返回 x 是否小于 y	(a<b)返回 True

续表

运算符	描　述	实例(a=10,b=20)
>=	大于等于,返回 x 是否大于等于 y	(a>=b)返回 False
<=	小于等于,返回 x 是否小于等于 y	(a<=b)返回 True

2.3.3 赋值运算符

Python 中的赋值运算符用来给对象赋值,其运算符和运算规则如表 2.3 所示。

表 2.3 赋值运算符和运算规则

运算符	描　述	实　例
=	简单的赋值运算符	c=a+b 将 a+b 的运算结果赋值为 c
+=	加法赋值运算符	c+=a 等效于 c=c+a
-=	减法赋值运算符	c-=a 等效于 c=c-a
*=	乘法赋值运算符	c*=a 等效于 c=c*a
/=	除法赋值运算符	c/=a 等效于 c=c/a
%=	取模赋值运算符	c %=a 等效于 c=c %a
**=	幂赋值运算符	c**=a 等效于 c=c**a
//=	取整赋值运算符	c//=a 等效于 c=c//a

2.3.4 逻辑运算符

Python 中的逻辑运算符和运算规则如表 2.4 所示。

表 2.4 逻辑运算符和运算规则

运算符	描　述	实例(x=True,y=False)
and	布尔“与”：仅当 x 为 True 且 y 为 True 时,x and y 返回 True,其他情况均返回 False	(x and y)返回 False
or	布尔“或”：仅当 x 为 False 且 y 为 False 时,x or y 返回 False,其他情况均返回 True	(x or y)返回 True
not	布尔“非”：如果 x 为 True,则返回 False。如果 x 为 False,则返回 True	not(x)返回 False

2.3.5　位运算符

位运算符是把数值转换成二进制再进行计算的。下面的变量 a 为 60,变量 b 为 13,位运算的运算原理如下:

```
a=0011 1100
b=0000 1101
-----------------
a&b=0000 1100
a|b=0011 1101
a^b=0011 0001
~a=1100 0011
```

Python 中的位运算符和运算规则如表 2.5 所示。

表 2.5　位运算符和运算规则

运算符	描　述	实例(a=60,b=13)
&	按位与运算符:参与运算的两个值。如果两个相应位都为 1,则该位的结果为 1,否则为 0	(a &b)输出结果为 12, 二进制解释:0000 1100
|	按位或运算符:只要对应的两个二进位有一个为 1 时,结果位就为 1	(a|b)输出结果为 61, 二进制解释:0011 1101
^	按位异或运算符:当两对应的二进位相异时,结果为 1	(a^b)输出结果为 49, 二进制解释:0011 0001
~	按位取反运算符:对数据的每个二进制位取反,即把 1 变为 0,把 0 变为 1	(~a)输出结果为-61, 二进制解释:1100 0011, 是一个有符号二进制数的补码形式
<<	左移运算符:运算数的各二进位全部左移若干位,由<<右边的数指定移动的位数,高位丢弃,低位补 0	a<<2 输出结果为 240, 二进制解释:1111 0000
>>	右移运算符:把>>左边的运算数的各二进位全部右移若干位,>>右边的数指定移动的位数	a>>2 输出结果为 15, 二进制解释:0000 1111

2.3.6　成员运算符

除了上述运算符之外,Python 还支持成员运算符,测试实例中包含了一系列的成员,包括字符串,列表或元组,如表 2.6 所示。

表 2.6 成员运算符和运算规则

运算符	描　述	实　例
in	如果在指定的序列中找到值则返回 True，否则返回 False	x 在 y 序列中，如果 x 在 y 序列中则返回 True
not in	如果在指定的序列中没有找到值则返回 True，否则返回 False	x 不在 y 序列中，如果 x 不在 y 序列中则返回 True

2.3.7 身份运算符

身份运算符用于比较两个对象的存储单元。下面实例中 id()函数用于获取对象的内存地址，如表 2.7 所示。

表 2.7 身份运算符和运算规则

运算符	描　述	实　例
is	is 是判断两个标识符是不是引用自一个对象	x is y，类似 id(x)==id(y)，如果引用的是同一个对象则返回 True，否则返回 False
is not	is not 是判断两个标识符是不是引用自不同对象	x is not y，类似 id(a) !=id(b)，如果引用的不是同一个对象则返回结果 True，否则返回 False

身份运算符的实例：

```
>>>x=10
>>>y=20
>>>x=y
>>>x is y
True
>>>#说明 x,y引用的是同一个数据对象
```

2.3.8 表达式

运算符与参与运算的对象一起构成了 Python 的表达式，表达式的运算要遵循运算符的优先级。表 2.8 列出了从最高到最低优先级的所有运算符。

表达式的运算实例，not "Abc"="abc" or 2+3<>5 and "23"<"3 "，运算结果如下：

表 2.8 表达式运算优先级

运 算 符	描 述
**	指数(最高优先级)
~、+、-	按位取反、加号和减号(最后两个的方法名为+@和-@)
*、/、%、//	乘、除、取模和取整除
+、-	加法和减法
>>、<<	右移、左移运算符
&	位运算符
^、|	位运算符
<=、<、>、>=	比较运算符
<>、==、!=	等于运算符
=、%=、/=、//=、-=、+=、*=、**=	赋值运算符
is、is not	身份运算符
in、not in	成员运算符
not、or、and	逻辑运算符

```
>>>not "Abc"=="abc" or 2+3!=5 and "23"<"3"
True
>>>
```

运算符优先级决定了运算符运算顺序,想要改变它们的计算顺序,可以使用圆括号。例如,not ("Abc"="abc" or 2+3<>5)and "23"<"3 "。



2.4 常用系统函数

2.4.1 常用内置函数

高级语言中的函数类似于数学中的函数,都是用来完成某些运算符无法完成的运算。Python 中的函数就是一段完成某个运算的代码的封装。

Python 中的函数分为内置函数、标准库函数和第三方库函数。丰富的库函数是

Python 区别于其他编程语言的主要特征，是快速问题求解和提高编程效率的主要工具。

1. 内置函数

Python 内置函数是指可以随着解释器的运行自动载入的函数，这类函数可以被随时调用，不需要使用语句进行导入，主要完成一些简单运算符无法实现的运算功能。

Python 内置函数按字母顺序的排列如表 2.9 所示。

表 2.9 Python 内置函数

abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Python 丰富的内置函数可以实现数值、字符、集合、输入/输出等对象的特定运算，本节只介绍其中几个常用的函数功能，其他函数会在后续章节中介绍。

2. 常用内置函数功能

常用内置函数的功能说明如表 2.10 所示。

表 2.10　Python 常用内置函数功能说明

函　数　名	功　　能
abs(x)	求绝对值,参数可以是整型,也可以是复数
all(iterable)	元素都为真时,函数值为 True;元素为空时,函数则返回 True
any(iterable)	元素有一个为真时,函数值为 True,元素为空时,函数则返回 False
bin(x)	将整数 x 转换为二进制数
bool([x])	将 x 转换为布尔型 Boolean 类型
chr(i)	返回整数 i 对应的 ASCII 字符
cmp(x, y)	如果 x<y ,则返回负数;如果 x==y,则返回 0;如果 x>y,则返回正数
complex([real[, imag]])	创建一个复数
dict([arg])	创建数据字典
dir([object])	返回当前范围内的变量或对象的详细列表
divmod(a, b)	返回一个由商和余数构成的元组,参数可以为整型或浮点型
eval(str [,globals[,locals]])	将字符串 str 当成有效的表达式求值并返回计算结果
float([x])	将一个字符串或数转换为浮点数。如果无参数则返回 0.0
format(value [, format_spec])	格式化输出字符串
help([object])	显示有关对象的帮助信息
hex(x)	将整数 x 转换为十六进制数
id(object)	返回对象的唯一标识
input([prompt])	返回用户输入
int([x[, base]])	将一个数字或 base 类型的字符串转换成整数,base 表示进制,默认为十进制
len(s)	返回集合长度
list([iterable])	生成一个列表
locals()	返回当前的变量列表
max(iterable[, args…][key])	返回集合中的最大值
min(iterable[, args…][key])	返回集合中的最小值
oct(x)	将一个数字转化为八进制数
open(name[,mode[,buffering]])	打开文件
pow(x, y[, z])	返回 x 的 y 次幂

续表

函 数 名	功 能
print(value1, value2…, sep=' ', end='\n')	按指定格式输出的函数
range([start], stop[, step])	产生一个序列,默认从 0 开始
round(x[, n])	将 x 进行四舍五入
sorted(iterable[, cmp[, key[, reverse]]])	对集合排序
str([object])	转换为字符串类型
sum(iterable[, start])	对集合求和
tuple([iterable])	生成一个元组
type(object)	返回该对象的类型

3. 内置函数应用实例

1) 数值运算函数

abs()函数:返回一个数值的绝对值,参数可以是整型或浮点型数据。

divmod(a,b):返回由 a/b 的商和余数构成的一个元组,a、b 可以是整型或浮点型。

pow(x, y[, z]):返回 x 的 y 次幂,如果有参数 z 则返回 x 的 y 次幂与 z 的模。

```
pow(x, y[, z])→pow(x, y)%z
pow(x, y)→x**y
```

round(x[, n]):对 x 进行四舍五入,n 为保留的小数位数,若默认则只保留整数。

```
>>>print(abs(10),abs(-10))
10 10
>>>print(divmod(10,4),divmod(10.5,2.5))
(2, 2) (4.0, 0.5)
>>>print(pow(2,3),pow(2,3,4))
8 0
>>>print(round(3.1415926,3),round(3.1415926))
3.142 3
>>>
```

2）数制转换函数

int([x[，base]])：将数值或字符串转换为十进制整数。

如果 x 是字符串，则它可能包含符号和小数点。参数 base 表示转换的基数（默认是十进制），它可以是[2，6]范围内的值或者 0。如果是 0，则系统将根据字符串内容解析。如果有参数 base，则参数 x 必须是一个字符串，否则将提示 TypeError 异常错误。

如果 x 是数值，则它可以是整数、长整数或浮点数。函数将 x 舍去小数部分，将其转换为整数。

x 是数值的情况：

```
>>>int(3.14)                #3
>>>int(2e2)                 #200
>>>int(100, 2)              #出错,有 base 参数时,函数只接收字符串
```

x 是字符串的情况：

```
>>>int('23',16)             #35,将十六进制数 23 转换为十进制整数 35
>>>int('Pythontab',8)       #出错,Pythontab 不是个八进制数
```

字符串 0x 视作为十六进制的符号，0b 视作为二进制的符号，0o 视作八进制的符号：

```
>>>int('0x10', 16)          #16,0x 是十六进制的符号
>>>int('0b10',2)            #2 ,0b 是二进制的符号
```

bin(x)：将十进制整数转换成二进制数。

hex(x)：将十进制整数转换成十六进制数。

oct(x)：将十进制整数转换成八进制数。

```
>>>b=255
>>>print(bin(b),hex(b),oct(b))
0b11111111 0xff 0o377
>>>a=0b10110011             #int()函数可以将二进制数转换成十进制
>>>print(int(a))
179
>>>print(int('123', 16),int('123',8),int('123'))
291 83 123
>>>
```

3）类型转换函数

bool([x])：返回 x 布尔值。

```
>>>x,y=0,1
>>>print(bool(x),bool(y))          0是 False,1是 True
False True
>>>print(bool('abc'),bool(''))     #字符串为 True,空串为 False
True False
>>>
```

float([x])：返回一个浮点数。

```
>>>float('+1.23')
1.23
>>>float(' -12345\n')
-12345.0
>>>float('1e-003')
0.001
>>>float('+1E6')
1000000.0
>>>float('-Infinity')
-inf
```

eval（str [,globals[,locals]]）：将字符串 str 当成有效的表达式求值并返回计算结果。

```
>>>x=1
>>>eval('x+1')
2
>>>a='5,10'
>>>x,y=eval(a)
>>>print(x,y)
5 10
>>>
```

4）集合运算函数

all()函数：元素均为 True 或元素为空时返回 True。

any()函数：元素至少一个为 True 时返回 True,元素为空时返回 False。

```
>>>print(all([1,2,3]),all([0,1,2]),all([]))
True False True
>>>print(any([1,2,3]),any([0,1,2]),any([]))
True True False
>>>
```

max(iterable[，args…][key])：返回集合元素中的最大值。

min(iterable[，args…][key])：返回集合元素中的最小值。

```
>>>mylist=[5,9,98,20,126]
>>>print(max(mylist),min(mylist))
126 5
>>>
```

range([start]，stop[，step])：产生一个序列,默认从 0 开始。

list([iterable])：产生一个列表。

tuple([iterable])：产生一个元组。

```
>>>x=range(1,10,2)              #生成一个 1<=x<10 的序列,步长为 2
>>>list(x)                      #由序列生成一个列表
[1, 3, 5, 7, 9]
>>>tuple(x)                     #由序列生成一个元组
(1, 3, 5, 7, 9)
>>>
```

5) 帮助函数

help([object])：查看函数用法的详细信息。

```
>>>help(round)                 #查看 round 函数的使用说明
Help on built-in function round in module builtins:
round(…)
round(number[, ndigits])->number

    Round a number to a given precision in decimal digits (default 0 digits).
    This returns an int when called with one argument, otherwise the
    same type as the number. ndigits may be negative.
>>>
```

6）输入/输出函数

（1）标准输入函数

input([prompt])：显示 prompt 提示信息，读取用户输入的数据，并返回一个字符串，提示信息 prompt 可以省略。

```
>>>s=input('input a data=>')
input a data=>Monty Python's Flying Circus
>>>print(s)
Monty Python's Flying Circus
>>>
```

input()函数返回的是输入数据的字符串，如果要输入数值数据，可以使用类型转换函数进行转换。例如：

```
>>>x=input('input x=>')
input x=>5
>>>y=input('input x=>')
input x=>10
>>>print(x+y)
510
>>>
>>>x=int(input('input x=>'))
input x=>5
>>>y=int(input('input x=>'))
input x=>10
>>>x+y
15
>>>
```

input()函数可以为多个变量赋值。例如：

```
>>>x,y=eval(input('input data to x,y=>'))
input data to x,y=>5,10
>>>print(x+y)
15
>>>
```

(2) 标准输出函数

print(value1, value2,…, sep=' ', end='\n'):其中,无参数时,函数表示输出一个空行;sep 表示插入在各输出值之间的分隔符,默认情况下以空格分隔;end 表示添加到最后一个输出值后面的结束符,默认情况下按 Enter 键结束(即换行输出)。

```
>>>print(10,20)                          #换行输出,空格分隔
10 20
>>>print(10,20,sep=',')                  #换行输出,逗号分隔
10,20
>>>print(10,20,sep=',',end='*')          #不换行输出,逗号分隔,*号结束
10,20*
>>>
```

(3) 格式化输出函数

format(value [, format_spec]):按照格式字符串的样式输出表达式的值,其中,根据不同的数据类型使用不同的格式字符。常用的格式字符如下:

d、b、o、x 或 X,分别表示输出十进制、二进制、八进制和十六进制整数。

f 或 F、e 或 E、g 或 G,分别表示按小数形式和科学记数法形式输出一个整数或浮点数。

c 表示输出以整数为编码的字符。

+、-、%分别表示输出正号、负号、百分号。

<、>、^分别表示输出字符左对齐、右对齐、居中对齐。

=表示填充字符位于符号和数字之间。

```
>>>x=65                                  #输出整数 x 的二进制,对应字符为十六进制
>>>print(format(x,'b'),format(x,'c'),format(x,'x'))
1000001 A 41
>>>y=3.1415926                           #输出浮点数,控制输出数据位数
>>>print(format(y,'f'),format(y,'e'),format(y,'g'))
3.141593  3.141593e+00 3.14159
>>>print(format(y,'.3f'),format(y,'6.3f'),format(y,'06.3f'))
3.142   3.142   03.142
>>>print(format(3.14,'+10'),format(3.14,'0=+10'))      #输出位数为 10 位
     +3.14+000003.14
>>>
```

2.4.2 常用标准库函数

Python 的标准库包含许多模块，每个模块中定义了很多有用的函数。与内置函数不同的是，Python 程序若要调用这些标准库函数，需要先使用 import 命令将函数或模块导入。常用的标准库有：数学库模块（math）、随机数模块（random）、时间（time）和日历（calendar）模块。

1. math 模块

math 模块提供数学相关的运算和函数，其中常用的函数如表 2.11 所示。

表 2.11 math 模块常用函数功能说明

函　　数	返　回　值
ceil(x)	对 x 的向上取整，如 ceil(4.1) 返回 5
exp(x)	返回 e 的 x 次幂，例如，exp(1)返回 2.718281828459045
fabs(x)	返回 x 的绝对值(浮点数)，如 fabs(-10) 返回 10.0
floor(x)	对 x 的向下取整，如 floor(4.9)返回 4
fmod (x,y)	返回 x/y 的余数(浮点数)，如 fmod(7,4)返回 3.0
log(x[,base])	返回 x 的自然对数，如 log(e)返回 1.0，可以用 base 参数改变对数的底，如 log(100,10)返回 2.0
log10(x)	返回 10 为基数的 x 的对数，如 log10(100)返回 2.0
max(x1, x2,…)	返回给定参数的最大值，参数可以为序列
min(x1, x2,…)	返回给定参数的最小值，参数可以为序列
pow(x, y)	返回 x 的 y 次幂，x**y 运算后的值
round(x [,n])	返回浮点数 x 的四舍五入值，如给出 n 值，则代表舍入到小数点后的位数
sqrt(x)	返回数字 x 的平方根，数字可以为负数，返回类型为实数，如 sqrt(4)返回 2.0

如果用 import 语句导入模块，则前面的模块名 math 不能省略，函数的调用格式如下：

```
math.<函数名>(<参数>)
```

函数的应用实例：

```
>>>import math
>>>print(math.fabs(-4),math.ceil(4.1),math.floor(4.9))
4.0  5 4
>>>print(math.pow(3,4),math.fmod(7,4),math.log(100,10))
81.0 3.0 2.0
```

2. random 模块

随机数可以用于数学、游戏以及安全等领域，还经常被嵌入到算法中，用以提高算法效率，并提高程序的安全性。Python 的常用随机数函数如表 2.12 所示。

如果用 from…import 语句导入模块，则前面的模块名 random 可以省略，函数的调用格式如下：

```
<函数名>(<参数 1>[,<参数 2>]…)
```

表 2.12　random 模块常用函数功能说明

函　　数	描　　述
choice(seq)	从序列的元素中随机挑选一个元素，如 choice(range(10))表示从 0 到 9 中随机挑选一个整数
randrange ([start,] stop [,step])	从指定范围内，按指定基数递增的集合中获取一个随机数，基数默认值为 1
random()	随机生成下一个实数，它在[0,1)范围内
sample(seq,k)	从序列中随机挑选 k 个元素
seed([x])	设置随机数生成器的种子，在调用随机函数前调用此函数。默认以系统时间作为种子
shuffle(lst)	将序列的所有元素随机排序
uniform(x, y)	随机生成[x,y]范围内的一个实数

函数的应用实例：

```
>>>from random import *          #导入 random 模块的所有函数
>>>x=[1,2,3,4,5]                 #x 赋值为一个序列
>>>sample(x,2)                   #从序列中随机挑选 2 个元素
[5, 1]
>>>shuffle(x)                    #将序列的元素随机排序
```

```
>>>x
[2, 5, 4, 3, 1]
>>>
```

3. time 模块

时间模块提供与时间有关的运算函数，例如返回当前系统时间和日期。常用的函数和功能说明如表 2.13 所示。

表 2.13 time 模块常用函数功能说明

函　数	描　述
time()	返回当前时间的时间戳(1970 纪元后经过的浮点秒数)
localtime([secs]	接收时间戳(1970 纪元后经过的浮点秒数)并返回一个时间元组 t(t.tm_isdst 可取 0 或 1，取决于当地当时是不是夏令时)
asctime([tupletime])	接收时间元组并返回一个日期时间字符串。时间元组省略时，返回当前系统时间。如“2008 年 12 月 11 日周二 18 时 07 分 14 秒”的 24 个字符的字符串
ctime([secs])	作用相当于 asctime(localtime(secs))，无参数时相当于 asctime()
strftime(fmt[,tupletime])	接收时间元组，按指定格式 fmt 返回当前日期和时间

函数应用实例：

```
>>>from time import *                          #导入 time 模块的所有函数
>>>time()                                      #返回当前时间的时间戳
1495318555.658038
>>>asctime()                                   #返回当前系统的日期和时间的字符串
'Sun May 21 06:16:11 2017'
>>>strftime('%Y-% m-%d %H:%M:%S')              #返回指定格式的当前日期和时间
'2017-05-21 06:33:48'
```

4. calendar 模块

日历模块(calendar)的函数都是与日历相关的，例如打印某月的字符月历。默认情况下，星期一是默认的每周第一天，星期日是默认的最后一天。如果需要更改默认设置，需要调用 calendar.setfirstweekday()函数。日历模块包含的函数如表 2.14 所示。

表 2.14　calendar 模块常用函数功能说明

函　　数	描　　述
setfirstweekday(weekday)	设置每周的第一天的日期码。默认为 0(星期一)到 6(星期日)
firstweekday( )	返回当前每周起始日期的设置。默认返回 0,即星期一
isleap(year)	如果 year 是闰年则返回 True,否则返回 False
leapdays(y1,y2)	返回在 y1 和 y2 两年之间的闰年总数
month(year,month,w=2,l=1)	返回指定年月的日历。多行字符串格式,两行标题,一周一行
monthcalendar(year,month)	返回一个整数列表。每个子列表代表一个星期
monthrange(year,month)	返回两个整数。第一个是该月的是星期几,第二个是该月的日期码
weekday(year,month,day)	返回指定日期的星期代码

函数应用实例:

```
>>>from calendar import *          #导入 calendar 模块的所有函数
>>>print(month(2017,5))            #多行格式输出 2017 年 5 月的日历
      May 2017
Mo Tu We Th Fr Sa Su
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
>>>print(weekday(2017,5,20))       #返回 2017 年 5 月 20 日的星期代码,5 代表星期六
5
>>>print(monthrange(2017,5))       #2017 年 5 月第一天是星期一,有 31 天
(0, 31)
>>>
```

习题 2

一、填空题

1. x=5 和 y=10,执行语句 x,y=y,x 后 x 的值是________。
2. x=5,执行语句 x-=2 之后,x 的值为________。
3. x=5,执行语句 x**=2 之后,x 的值为________。

4. 表达式 int('11',16) 的值为________。

5. 表达式 int('11',8) 的值为________。

6. 表达式 int('11',2)的值为________。

7. 语句 print(10,20,30，sep=':') 的输出结果为________。

8. 内置函数________用来返回序列中的最大元素。

9. 内置函数________用来返回序列中的最小元素。

10. 内置函数________用来返回数值型序列中所有元素之和。

11. 查看变量内存地址内置函数是________。

12. 表达式 1<2<3 的值为________。

13. 表达式 3|5 的值为________。

14. 表达式 3&6 的值为________。

15. 表达式 3<<2 的值为________。

16. 表达式 3**2 的值为________。

二、判断题

1. 在 Python 中可以使用 for 作为变量名。 (　　)

2. Python 关键字不可以作为变量名。 (　　)

3. Python 变量名区分大小写,所以 student 和 Student 不是同一个变量。 (　　)

4. 假设 random 模块已导入,那么表达式 random.sample(range(10)，7)的作用是生成 7 个不重复的整数。 (　　)

5. Python 用运算符“//”计算除法。 (　　)

6. 转义字符“\n”的含义是换行符。 (　　)

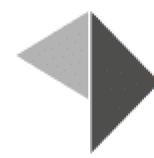
7. 标准 math 库中用来计算幂的函数是 sqrt()。 (　　)

第 3 章

Python 控制语句

学习目标

- 理解结构化程序设计的三种基本结构
- 掌握分支结构 if 语句的用法
- 掌握 for 语句循环结构
- 掌握 while 语句循环结构



3.1 结构化程序设计

结构化程序设计方法的基本思想是以系统的逻辑功能设计和数据流关系为基础，根据数据流程图和数据字典，借助于标准的设计准则和图表工具，通过“自上而下”和“自下而上”的反复，逐层把系统划分为多个大小适当、功能明确、具有一定独立性并容易实现的模块，从而把复杂系统的设计转变为多个简单模块的设计。结构化程序设计方法可以用三个词语进行概括：自上而下、逐步求精、模块化设计。

结构化的含义是指用一组标准的准则和工具从事某项工作。在结构化程序设计之前，每一个程序员都按照各自的习惯和思路编写程序，没有统一的标准，也没有统一的技术方法，因此，程序的调试、维护都很困难，这是造成软件危机的主要原因之一。20 世纪 60 年代，计算机科学家提出了有关程序设计的新理论，即结构化程序设计理论，该理论认为，任何一个程序都可以用三种基本逻辑结构编制，而且只需这三种结构。这三种结构分别是顺序结构、分支结构(选择结构)和循环结构。这种程序设计的新理论，促使人们采用模块化编制程序，把一个程序分成若干个功能模块，这些模块之间尽量彼此独立，用控制语句或过程调用语句将这些模块连接起来，形成一个完整的程序。一般来说，结构化程序设计方法不仅大大改进了程序的质量和程序员的工作效率，而且还增强了程序的可读性和可维护性。

计算机程序一般都由三种基本结构组成：顺序结构、分支结构(选择结构)和循环结构。结构化程序的结构可以用多种方式表示，例如程序流程图、N-S 图、PAD 图等。程序流程图是广泛使用的结构化设计表示工具，具有表达直观，易于掌握的特点。

3.1.1 顺序结构

顺序结构是指程序从第一行语句开始执行，执行到最后一行语句结束，程序中的每条语句都会被执行一次。顺序结构的程序流程图如图 3.1 所示。

3.1.2 分支结构

分支结构也称选择结构，表示程序的处理步骤出现了分支，它需要根据某一特定的条件选择其中的一个分支执行。分支结构有单分支、双分支和多分支三种形式。

1. 单分支结构

当判断条件为真值时，执行语句块 1；当判断条件为假值时，越过语句块往下执行其他语句或结束，通常用来指定某一段语句是否执行。单分支结构的程序流程图如图 3.2 所示。

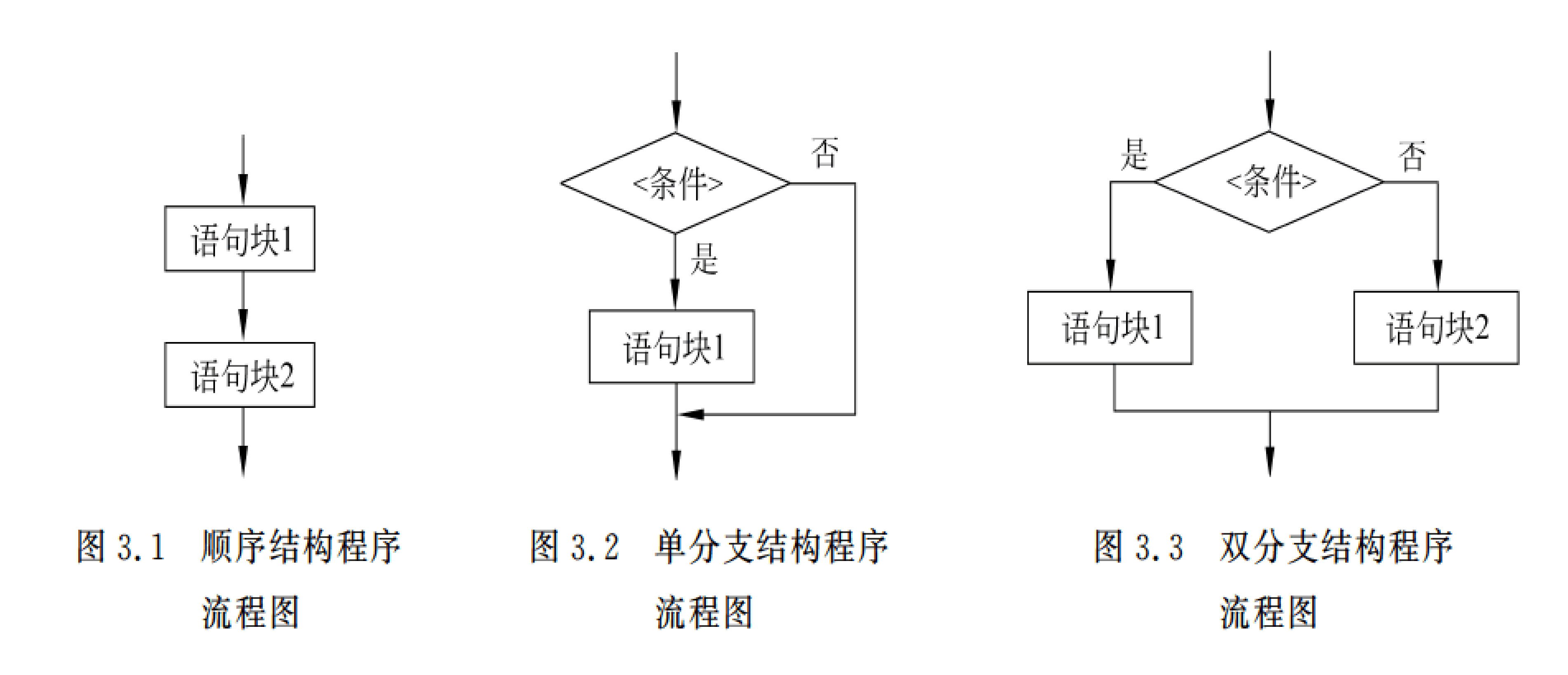


图 3.1 顺序结构程序流程图　　图 3.2 单分支结构程序流程图　　图 3.3 双分支结构程序流程图

2. 双分支结构

当判断条件为真值时，执行语句块 1；当判断条件为假值时，执行语句块 2，通常用来有选择地在两段语句中选择一段执行。双分支结构的程序流程图如图 3.3 所示。

3. 多分支结构

多分支结构即扩展的双分支结构，一般设有 n 个条件，n 或者 $n+1$ 个语句块，当判断

条件从上向下判断到某个为真值时,执行对应的语句块,然后退出多分支结构继续向下执行其他内容。需要注意的是,多分支结构在一次执行时只能选择一个分支,即使其他判断条件为真值,也不会继续判断执行。多分支结构的程序流程图如图3.4所示。

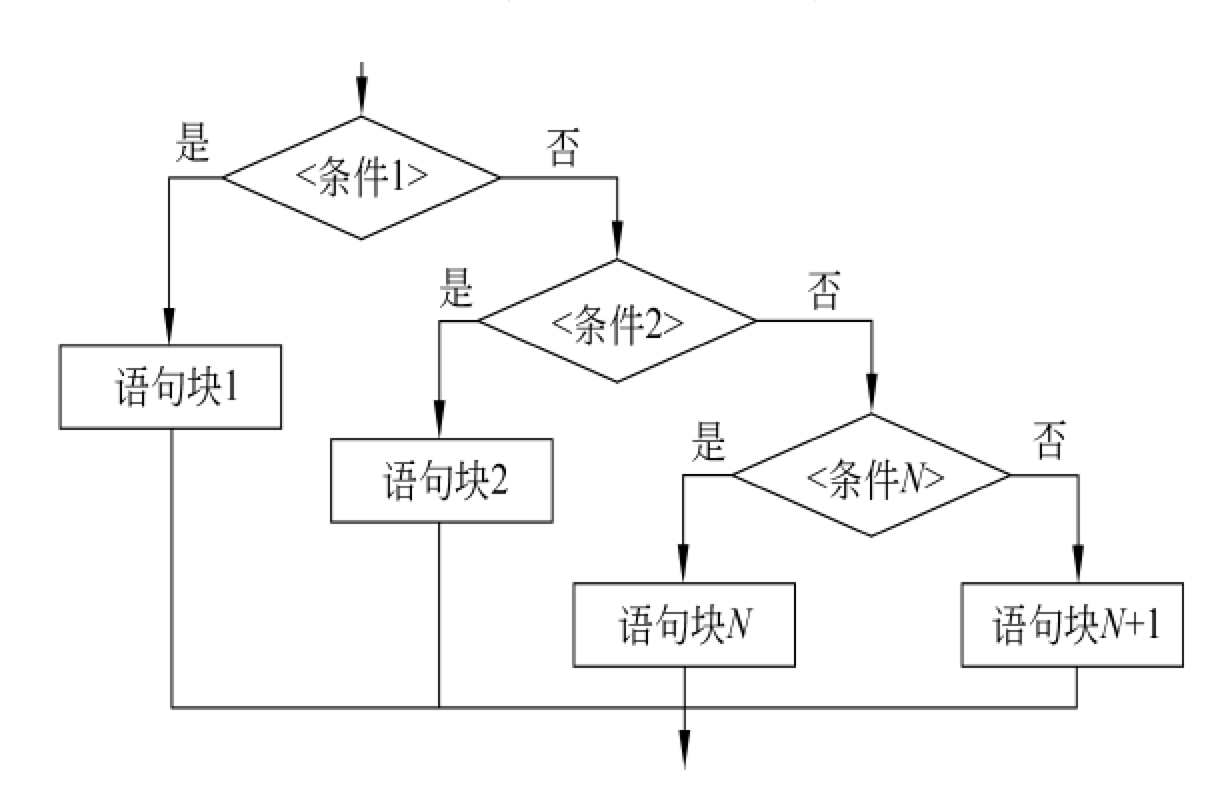


图3.4 多分支结构程序流程图

3.1.3 循环结构

循环结构是程序根据条件判断,在满足条件的情况下反复执行某个语句块的运行方式。Python中根据循环触发条件的不同,分为条件循环和遍历循环。条件循环结构在执行循环时先判断循环条件的取值,如果为True则执行一次语句块(循环体),然后返回继续判断循环条件,如果循环条件为False,则结束循环。条件循环的程序流程图如图3.5所示。遍历循环在执行循环时也要进行判断,看循环变量是否在遍历队列中,如果在则取一个遍历元素,执行一次语句块(循环体),然后返回再取下一个遍历元素,直到遍历元素取完,循环结束。遍历循环的程序流程图如图3.6所示。

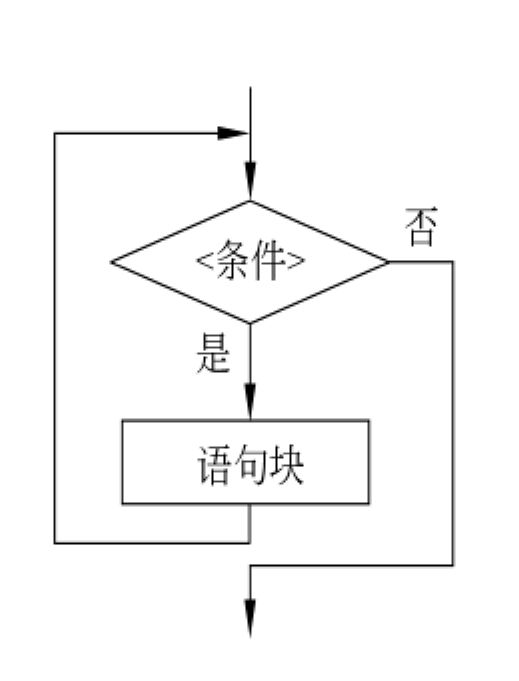


图3.5 条件循环结构程序流程图

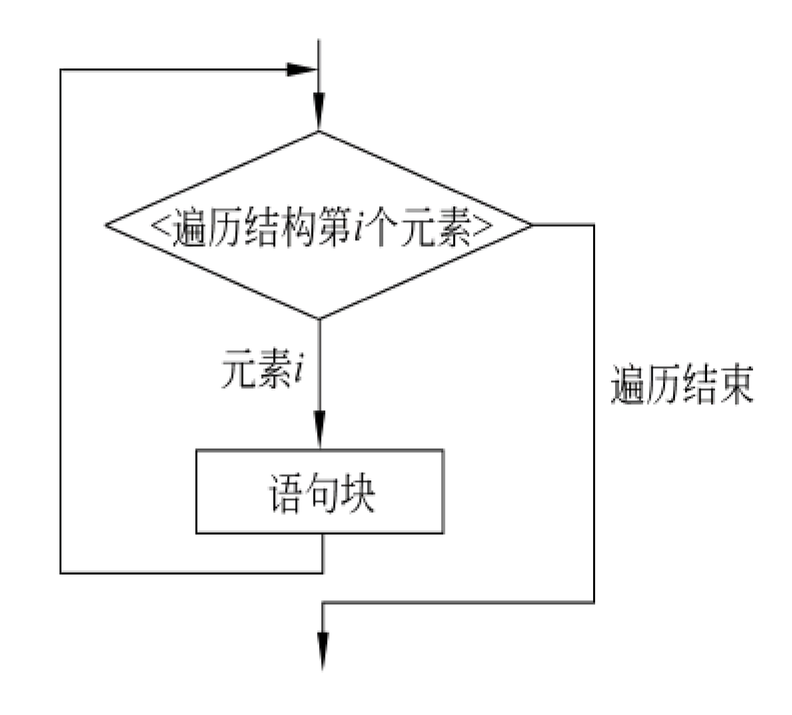


图3.6 遍历循环结构程序流程图



3.2 分支结构

3.2.1 单分支结构

Python 中单分支 if 语句的语法格式如下：

```
if  <条件>:
<语句块>
```

条件是一个表达式，其结果一般为真值 True 或者假值 False。语句块是 if 条件满足后执行一条或多条语句的序列。程序执行到 if 语句时，如果判断到条件为 True，则执行语句块；若条件为假，则跳过语句块。不管条件为 True 还是 False，单分支执行结束后都会执行与 if 语句同级别的下一条语句继续执行程序。

分支结构中的条件在多数情况下是一个关系比较运算。形成比较的最常见方式是利用关系操作符。Python 语言共有 6 个关系操作符，如表 2.2 所示。

【例 3.1】 输入两个数字，输出其中较大的数字。

```
#example3.1
x=eval(input("请输入第一个数字:"))
y=eval(input("请输入第二个数字:"))
print("输入的两个数字为: ",x,y)
if x>y:
    print("较大的是:",x)
if x<y:
    print("较大的是:",y)
```

程序运行结果如下(输入 8 和 15)：

```
>>>
===============RESTART:C:/Python/python36/example3.1.py============
请输入第一个数字:8
请输入第二个数字:15
输入的两个数字为: 8 15
较大的是: 15
>>>
```

说明：

(1) 可以用 input()函数输入两个数字，然后对它们进行判断比较，如果某个数字比较大，则用 print()函数输出该数字。

(2) 运行后，程序对输入的两个数字进行判断，例如当输入的两个数字为 8 和 15 时，执行第一个 if 语句，判断条件 x>y 为假值 False，不执行 print("较大的是：",x)语句，分支判断结束；执行第二个 if 语句，判断条件 x<y 为真值 True，执行 print("较大的是：",y)语句，所以程序显示了“较大的是：15”。注意：如果输入的两个数字相等，由于该程序没有考虑到这种情况，则没有显示输出结果。

3.2.2 双分支结构

双分支结构是使用比较多的一种程序结构，Python 中双分支 if 语句的语法格式如下：

```
if <条件>:
    <语句块 1>
else:
    <语句块 2>
```

双分支结构根据条件的真假值有选择地执行语句块 1 或语句块 2。当条件为 True 时，执行语句块 1，然后结束分支结构；当条件为 False 时，执行语句块 2，然后结束分支结构。

【例 3.2】 用双分支结构改写例 3.1。

```
#example3.2
x=eval(input("请输入第一个数字:"))
y=eval(input("请输入第二个数字:"))
print("输入的两个数字为: ",x,y)
if x>y:
    print("较大的是:",x)
else:
    print("较大的是:",y)
```

程序运行结果如下(输入顺序为 10 和 5)：

```
>>>
=========RESTART: C:/ Python/Python36/example3.2.py=========
```

```
请输入第一个数字:10
请输入第二个数字:5
输入的两个数字为: 10 5
较大的是: 10
>>>
```

说明：

(1) 如果不考虑两个数字相等的情况，可以将例3.1中的两个单分支if结构合并为一个双分支结构。

(2) 运行后，如果x>y，则执行if结构下的print("较大的是:",x)语句；否则执行else结构下的print("较大的是:",y)语句。注意：如果输入的两个数字相等，使用此结构和例3.1的输出结果不同，因为使条件判断x>y为假的情况可能是x<y，也可能是x=y，所以一旦输入的两个数字相同，程序将显示“较大的是：y”的值。

3.2.3 多分支结构

多分支结构是对双分支结构的一种补充，当判断的条件有多个且判断结果有多个时，可用多分支if语句进行判断。多分支结构语法格式如下：

```
if  <条件 1>:
    <语句块 1>
elif<条件 2>:
    <语句块 2>
elif<条件 3>:
    <语句块 3>
…
else:
    <语句块 n>
```

程序执行时，会按照条件 *n* 的序列从上向下进行判断，当第一个条件i的值为True，就执行该条件下的语句块，然后整个多分支if结构结束。如果没有任何条件为True，则执行else下的语句块。注意：else是可选的。

【例 3.3】 用多分支结构改写例3.1，如果两个数字相等，也要给出说明。

```
#example3.3
x=eval(input("请输入第一个数字:"))
y=eval(input("请输入第二个数字:"))
```

```
print("输入的两个数字为：",x,y)
if x>y:
    print("较大的是:",x)
elif x<y:
    print("较大的是:",y)
else:
    print("这两个数字相等")
```

说明：

(1) 两个数字相互比较，只能有三种结果，即大于、小于、相等。

(2) 本例中有两个判断，在 x>y 和 x<y 这两种情况都不成立时，会执行 else 下的语句 print("这两个数字相等")。在多分支结构中，如果有多个判断都为 True，也只会执行第一个判断为 True 的分支中的语句块，执行后整个多分支结构结束，所以，哪个条件写在上，哪个条件写在下，有时会影响程序执行的结果。

【例 3.4】 输入一个分数，判断它对应到的学分绩点。90 分以上绩点为 4；80～90 分绩点为 3；70～79 分绩点为 2；60～69 分绩点为 1；60 以下绩点为 0。请问：以下两段代码哪一段是正确的。

程序 1 代码如下：

```
#example3.41
score=eval(input("请输入分数："))
if score>=90:
    gpa=4
elif score>=80:
    gpa=3
elif score>=70:
    gpa=2
elif score>=60:
    gpa=1
else:
    gpa=0
print("应得学分绩点为：",gpa)
```

程序 2 代码如下：

```
#example3.42
score=eval(input("请输入分数："))
```

```
if score<60:
    gpa=0
elif score>=60:
    gpa=1
elif score>=70:
    gpa=2
elif score>=80:
    gpa=3
else:
    gpa=4
print("应得学分绩点为:",gpa)
```

程序 1 运行结果如下(输入 85):

```
>>>
=======RESTART: C:/Python/Python36/example3.41.py=======
请输入分数:85
应得学分绩点为:3
>>>
```

程序 2 运行结果如下(输入 85):

```
>>>
=======RESTART: C:/ Python/Python36/example3.42.py=======
请输入分数:85
应得学分绩点为:1
>>>
```

说明:

(1) 经过比较可以看出,虽然两段程序都可以顺利执行,但是程序 2 并不符合题目的要求。

(2) 例如在输入 85 时,程序 1 可以得出 gpa 为 3,而程序 2 得到的 gpa 只有 1,原因就是在程序 2 中,进行 score>60 的判断结果为 True 时,将 gpa 赋值为 1,然后就结束了整个多分支结构。

使用多分支 if 语句时,一定要注意思路清晰,养成良好的程序书写风格,层次明确,便于阅读和修改程序。

3.2.4 分支结构的嵌套

如果一个 if 分支结构中包含另一个(或多个)if 分支,则称为分支结构的嵌套。

【例 3.5】 输入 3 个数字,利用分支结构对其进行降序排列输出。

```
#example3.5
a=eval(input("输入第 1个数字"))
b=eval(input("输入第 2个数字"))
c=eval(input("输入第 3个数字"))
print("输入顺序为:",a,b,c)
if a<b:
    a,b=b,a
if a<c:
    print("排序后为:",c,a,b)
else:
    if c>b:
        print("排序后为:",a,c,b)
    else:
        print("排序后为:",a,b,c)
```

程序运行结果如下:

```
>>>
=======RESTART: C: /Python/Python36/example3.5.py =======
输入第 1个数字 5
输入第 2个数字 3
输入第 3个数字 4
输入顺序为: 5 3 4
排序后为: 5 4 3
>>>
```

说明:

(1) 先比较输入的前两个数字 a 和 b 的大小,如果 a 小于 b,则交换 a 和 b 的顺序,第一个单分支结构保证了 a 一定要大于 b。

(2) 在双分支结构中,如果 a<c,那么 c 一定是最大的,按照 c、a、b 的顺序输出排序结果。如果 a>c,则再用一个双分支判断 b 和 c 的大小,如果 b>c,则按照 a、b、c 的顺序输出排序结果;如果 b<c,则按照 a、c、b 的顺序输出排序结果。



3.3 循环结构

循环结构是根据条件重复执行某些语句,它是程序设计中的一种重要结构。使用循环控制结构可以减少程序中的大量重复语句,从而编写出更简洁的程序。Python 提供了两种不同风格的循环结构,包括遍历循环 for 语句和条件循环 while 语句。

一般情况下,for 语句循环按给定的次数进行循环,而 while 语句循环是在条件满足时执行循环。

3.3.1 for 语句循环

遍历循环可以理解为让循环变量逐一使用一个遍历结构中的每个项目,遍历结构可以是字符串、列表、文件、range()函数等。for 语句循环语法格式如下:

```
for<循环变量>in<遍历结构>:
    <语句块>
```

for 语句的执行过程是:每次循环,判断循环变量值是否还在序列中,如果在,则取出该值提供给循环体内的语句使用;如果不在,则结束循环。

【例 3.6】 求 1 到 100 之间的奇数和。

程序 1 代码如下:

```
#example3.61
s=0
for i in range(1,101,2):
    s=s+i
print("1到 100之间的奇数和为:",s)
```

程序运行结果如下:

```
>>>
=======RESTART: C: /Python/Python36/example3.61.py=======
1到 100之间的奇数和为: 2500
>>>
```

程序 2 代码如下:

```
#example3.62
s=0
for i in range(1,101):
    if i%2==1:
        s=s+i
print("1到100之间的奇数和为:",s)
```

程序运行结果如下：

```
>>>
=======RESTART: C:/Python/Python36/example3.62.py=======
1到100之间的奇数和为:2500
>>>
```

说明：

(1) 程序 1 定义一个初始值为 0 的变量 s，然后通过 range()函数产生 1 到 100 之间的奇数值，利用循环变量取遍这些值，然后将它们加到 s 中。

(2) 注意：range()函数可以产生某范围内的整数，如果只有一个参数，例如 range(5)，产生从 0 到 5 之间的整数，包括 0 但是不包括 5，即[0,4]，如果有两个参数，例如 range(3,7)，产生的数字范围为[3,6]；如果有三个参数，则第三个参数代表步长值，即从初值到终值过渡时每次加的数字，例如 range(5,12,2)产生的数字列表为[5,7,9,11]，range()函数也可以产生一个由大到小的数字列表，但是步长参数要为负数，例如 range(10,5,-1)产生的数字为[10,9,8,7,6]。

(3) 程序 1 中，for 语句执行时，range(1,101,2)函数产生遍历结构为[1,100]并且由 1 开始每次加数字 2，直到小于 101 结束，而循环变量 i 每次遍历其中一个数字，通过 s=s+i 语句累加到变量 s 中。

(4) 程序 2 定义一个初始值为 0 的变量 s，然后通过 range()函数产生 1 到 100 之间所有的整数，利用循环变量取遍这些值，再通过分支语句 if 判断这些值是不是奇数，如果是奇数，就加到 s 中。

(5) 程序 2 中，range(1,101)只负责产生[1,100]之间的整数，i 遍历取遍这些数字，由 if 语句判断这些数字中哪些是奇数，如果满足 i%2==1 这个条件，就累加到 s 中，虽然这段代码用了循环嵌套分支的结构，相比程序 1 复杂，但是这段代码更具有通用性，可以解决的问题种类更多。

for 语句循环还可以使用 else 关键字，其语法格式如下：

```
for<循环变量>in<遍历结构>:
    <语句块 1>
else:
    <语句块 2>
```

在这种结构中,当 for 循环正常执行后,程序会继续执行 else 语句中的内容。如果 for 循环因为某种原因没有正常执行完,如遇到了 break 语句,则不会执行 else 语句中的内容。所以通常用 else 检验 for 循环是否结束。例 3.6 中的程序 2 也可以改为以下格式:

```
#example3.62
s=0
for i in range(1,101):
    if i%2==1:
        s=s+i
else:
    print("计算完毕。",end="")
print("1到100之间的奇数和为:",s)
```

程序运行结果如下:

```
>>>
=======RESTART: C:/Python/Python36/example3.62.py=======
计算完毕。1到100之间的奇数和为:2500
>>>
```

需要注意的是,在使用 else 的循环中,else 语句要和 for 对齐,而不是和 for 循环中的 if 语句对齐。

【例 3.7】 求字符串“Life is short, YOU need Python!”中有多少个字母“o”,不区分大小写字母。

```
#example3.7
n=0
str="Life is short, YOU need Python!"
for i in str:
    if i=="o" or i=="O":
        n=n+1
else:
```

```
    print("计算完毕。",end="")
print("字母 o 的个数为:",n)
```

代码执行结果如下：

```
>>>
=======RESTART: C: /Python/Python36/example3.7.py=======
计算完毕。字母'o'的个数为: 3
>>>
```

说明：先设置一个初始值为 0 的变量 n，作为计数器，然后将字符串作为一个遍历结构，循环变量 i 会每次取字符串中的一个字母，再判断该字母是不是“o”，如果是，便将 n 增加 1。

3.3.2 while 语句循环

在明确知道循环的次数或者明确遍历结构时，一般使用 for 循环。但是更多时候无法明确遍历结构，或者不确定循环需要进行多少次。这时，就要使用 while 循环了。while 语句循环语法格式如下：

```
while<条件>:
    <语句块>
```

其中<条件>结果为 True 或者 False。当条件为 True，则执行一遍语句块，然后返回到 while 语句继续判断<条件>；当条件为 False 时循环结束。

和 for 一样，while 循环也有一种带有 else 的表达形式，语法格式如下：

```
while<条件>:
    <语句块 1>
else:
    <语句块 2>
```

在这种结构中，当 while 循环正常结束后，程序会继续执行 else 语句中的语句块 2，一般用来检验 while 循环是否结束。

【例 3.8】 猜价格。首先产生一个[10，100]之间的随机整数作为价格并赋予一个变量，如 n。然后用户可以输入数字猜价格，如果输入的数字大于 n 或者小于 n，则给予用户相应的提示；如果猜对了，则告诉用户猜中了。

```
#example3.8
from random import randint
n=randint(10,100)
print("商品价格已经产生,请输入 10 到 100 间的价格:")
bingo=False
while bingo==False:
    guess=eval(input("请输入您猜的价格:"))
    if guess>n:
        print("您输入的价格高于指定价格,请继续。")
    elif guess<n:
        print("您输入的价格低于指定价格,请继续。")
    else:
        print("恭喜您猜对了!价格为",guess)
        bingo=True
else:
    print("游戏结束!")
```

程序运行的结果如下:

```
>>>
=======RESTART: C: /Python/Python36/example3.8.py=======
商品价格已经产生,请输入 10 到 100 间的价格。
请输入您猜的价格:80
您输入的价格低于指定价格,请继续。
请输入您猜的价格:90
您输入的价格高于指定价格,请继续。
请输入您猜的价格:85
您输入的价格低于指定价格,请继续。
请输入您猜的价格:88
您输入的价格高于指定价格,请继续。
请输入您猜的价格:87
恭喜您猜对了!价格为 87
游戏结束!
>>>
```

说明:

(1) 产生一个随机整数,可以引用函数库 random 中的 randint(m,n)函数,参数 m<=n,函数会产生[m,n]之间的随机整数。

(2) 当程序进入 while 循环时，判断条件 bingo=False 为真值，意味着开始循环。循环中使用 input()函数为变量 guess 赋值，然后用多分支 if 对 guess 进行判断，当 guess 的值大于或者小于 n，都给予文字提示，然后分支结束，返回 while 语句继续循环；当 guess 的值等于 n，多分支结构 if 结构执行 else 下的语句：输出“恭喜您猜对了！”，并且把 bingo 变量赋值为 True，分支结束。当返回 while 语句时，bingo=False 就变成了假值，循环结束，执行 while 语句同层的 else 下的语句。

(3) 本例中，循环是依靠一个条件“输入价格不等于初始产生的价格”进行的，由于用户输入的数字次数，也就是猜价格的次数未知，所以不能采用 for 语句循环结构。而 while 语句循环结构就可以处理这种未知循环次数的循环。

当然，while 语句循环也可以用来编写已知循环次数的循环。用 while 循环构造一个数字范围，一般来说是在 while 语句之前定义循环变量的初始值；在循环语句中指定循环变量的步长值；在 while 语句的条件处设置循环的终止值。

【例 3.9】 用 while 循环求 100 以内的奇数和。

```
#example3.9
s=0
i=1
while i<=100:
    if i%2==1:
        s=s+i
    i+=1
else:
    print("计算完毕。",end='')
print ("1到100之间的奇数和为:",s)
```

程序运行结果如下：

```
>>>
=======RESTART: C: /Python/Python36/example3.9.py=======
计算完毕。1到100之间的奇数和为: 2500
>>>
```

说明：i=1 的作用是定义循环变量的初始值，在 while 条件处设置 i<=100 为循环变量的终止值，在循环中 i+=1 语句和 i=i+1 等价，设置每次 i 增加的步长。

在编写 while 语句循环时，如果条件的计算结果一直为 True，而循环中没有 break 结束 while 循环，也就是没有逻辑出口，则循环将陷入永远执行的状态，称为死循环。例如

以下两行语句组成的程序,将一直输出数字 1。

```
while True:
    print(1)
```

在例 3.9 中,如果删除 i=i+1 语句,i 值在每次循环时都是 1,而循环条件是 i<=100,每次判断时都为 True,循环将始终执行。如果程序陷入死循环,在 IDLE 环境中可以按组合键 Ctrl+C,开发环境中会显示“KeyboardInterrupt”,程序终止。

3.3.3 循环的嵌套

在循环语句中使用另一个循环语句称为循环的嵌套,也称多重循环。for 语句循环和 while 语句循环都可以互相嵌套。利用循环的嵌套可以实现更复杂的程序设计。例如有如下代码段:

```
for i in range(1,4):
    for j in range(1,4):
        print("i 值为",i,";","j 值为",j)
```

可以看到,程序段的运行结果如下:

```
i 值为 1 ; j 值为 1
i 值为 1 ; j 值为 2
i 值为 1 ; j 值为 3
i 值为 2 ; j 值为 1
i 值为 2 ; j 值为 2
i 值为 2 ; j 值为 3
i 值为 3 ; j 值为 1
i 值为 3 ; j 值为 2
i 值为 3 ; j 值为 3
```

上层的 i 循环称为外层循环,下层的 j 循环称为内层循环,当外层循环执行一次时,内层循环就要整体循环一遍。从上面程序的结果中可以看出,当外层循环的循环变量 i 为 1 时,内层循环就要完成循环变量 j 从 1 到 3 的取值,当内层循环结束后,再次返回外层循环,i 值变为 2,内层循环再一次完整循环一次,以此类推。如果把两个循环加上 else 语句,则程序代码如下:

```
for i in range(1,4):
    for j in range(1,4):
        print("i值为",i,";","j值为",j)
    else:
        print("内层循环结束。")
else:
    print("外层循环结束。")
```

程序的运行结果如下：

```
i值为 1 ; j值为 1
i值为 1 ; j值为 2
i值为 1 ; j值为 3
内层循环结束。
i值为 2 ; j值为 1
i值为 2 ; j值为 2
i值为 2 ; j值为 3
内层循环结束。
i值为 3 ; j值为 1
i值为 3 ; j值为 2
i值为 3 ; j值为 3
内层循环结束。
外层循环结束。
```

可以看出，外层循环结束一次，而内层循环一共结束了 3 次。

【例 3.10】 解中国古代一道著名的数学题——百元百鸡问题。已知：公鸡 5 元 1 只；母鸡 3 元 1 只；小鸡 1 元 3 只。问要想 100 元正好买 100 只鸡，应该如何购买？

```
#example3.10
for x in range(1,21):
    for y in range(1,34):
        z=100-x-y
        if 5*x+3*y+z/3==100:
            print("公鸡",x,"母鸡",y,"小鸡",z)
else:
    print("计算完毕")
```

程序运行结果如下：

```
>>>
=======RESTART: C: \Python\Python36\例题 3.10.py=======
公鸡 4 母鸡 18 小鸡 78
公鸡 8 母鸡 11 小鸡 81
公鸡 12 母鸡 4 小鸡 84
计算完毕
>>>
```

说明：

(1) 如果用列方程的方法解决，可以知道只能根据鸡的数量等于 100 或者钱的数量等于 100 列出两个方程，而问题中有 3 个未知数，无法求得具体的值，在计算机中，可以采用穷举法求解，即对所有的可能解，逐个进行试验，若满足条件，就得到一组解，否则继续测试，直到循环结束为止。

(2) 假设公鸡有 x 只，则 100 元全部买公鸡的话，最多可以买 20 只，如果最少需要买 1 只公鸡，那么 x 的取值范围为 range(1,21)；假设母鸡有 y 只，则 100 元全部买母鸡的话，最多可以买 33 只，如果母鸡最少需要买 1 只，那么 y 的取值范围为 range(1,34)，利用双重循环组合 x 和 y 的值，当 x 和 y 确定为某一个数值后，小鸡的数量如果用 z 代表，则 z 的值为 $100-x-y$，每次循环都测试条件 $5*x+3*y+z/3=100$ 是否成立，如果条件成立，则找到一组合适的解。

和分支结构的嵌套一样，编写循环的嵌套时，要注意语句或者语句段的所属层次，认清是外层循环中的语句还是内层循环中的语句。



3.4 break 语句和 continue 语句

3.4.1 break 语句

Python 提供了一个提前结束循环的语句——break 语句。在循环中，执行到 break 语句时，可以结束本层的循环。一般来说，break 语句要放在一个分支结构中，当触发某个条件时，结束循环的运行。例如有如下代码：

```
for s in "python":
    for i in range(1,4):
        print(s,end="")
```

作用是把“python”中的每个字母都重复三遍，不换行输出。程序运行结果如下：

```
>>>
=======RESTART: C:/Python/Python36/temp.py=======
pppyyytttthhhooonnn
>>>
```

修改代码如下：

```
for s in "python":
    for i in range(1,4):
        if s=="h":
            break
        print(s,end="")
```

程序运行结果如下：

```
>>>
=======RESTART: C:/Python/Python36/temp.py=======
pppyyytttooonnn
>>>
```

说明：

(1) 在内层循环中，如果 s 等于字母 h，则跳出内层循环。但是外层循环将继续运行，程序依然会输出字母 h 以后的字母 o 和字母 n。

(2) 使用 break 时，要注意这条语句属于哪个循环。

【例 3.11】 求两个数字的最小公倍数。

```
x=eval(input("输入第一个数字"))
y=eval(input("输入第二个数字"))
if x<y:
    x,y=y,x
for i in range(x,x* y+1):
    if i%x==0 and i%y==0:
        print(x,"和",y,"的最小公倍数为",i)
        break
```

程序运行结果如下：

```
>>>
=======RESTART: C: /Python/Python36/example3.11.py=======
```

```
输入第一个数字 5
输入第二个数字 10
10和 5的最小公倍数为 10
>>>
```

说明：

(1) 程序运行后，可以通过 input()函数输入两个数字给变量，例如 x 和 y，假设 x 大于 y。当 x 和 y 互质时，x 和 y 的最小公倍数为 $x*y$；当 x 能被 y 整除时，x 和 y 的最小公倍数为 x，即 x 和 y 的最小公倍数在区间[x,$x*y$]之间。

(2) 首先利用 if 分支判断 x 和 y 的大小，如果 $x<y$ 则交换 x 和 y 的值，这样保证了 $x>=y$。在 for 语句循环中，如果循环变量 i 能够被 x 和 y 整除，则 i 是 x 和 y 的公倍数，而不一定是最小公倍数，如当输入两个不互质的数字 10 和 5 时，如果代码省略了 break 语句，则程序运行结果如下：

```
>>>
=======RESTART: C: /Python/Python36/example3.11.py=======
10 和 5 的最小公倍数为 10
10 和 5 的最小公倍数为 20
10 和 5 的最小公倍数为 30
10 和 5 的最小公倍数为 40
10 和 5 的最小公倍数为 50
>>>
```

当使用了 break 时，循环变量 i 遍历到第一个满足 if 分支的数字，输出计算结果后循环就结束，程序只能输出一个数字，并且是最小公倍数。

3.4.2 continue 语句

continue 语句和 break 语句一样，用在循环结构中，用来结束循环的运行，但是 continue 语句只能结束本次循环的执行，而不终止循环。即当执行到 continue 语句时，程序会终止当前循环，并忽略循环中 continue 之后的语句，然后回到循环语句 for 或者 while，再次判断是否进行下一次循环。

【例 3.12】 输入 10 名同学的分数求及格同学的均值，如果分数低于 60，则不计入计算中。

程序 1 代码如下：

```
#example3.121
s,n=0,0
for i in range(1,11):
    score=eval(input("输入成绩"))
    if score>=60:
        s=s+score
        n=n+1
print("合格人数为:",n)
print("成绩平均值为:",round(s/n,2))
```

程序运行结果如下:

```
>>>
=======RESTART: C: /Python/Python36/example3.121.py=======
输入成绩 70
输入成绩 80
输入成绩 90
输入成绩 60
输入成绩 70
输入成绩 80
输入成绩 90
输入成绩 34
输入成绩 56
输入成绩 67
合格人数为: 8
成绩平均值为: 75.88
>>>
```

说明:

(1) 首先设置初始变量 s 代表总分,n 代表合格人数。

(2) 循环 10 次,采用 if 分支结构进行判断,如果输入的 score 大于等于 60 分,则进行累加并且求 60 分以上的人数。循环结束之后,输出计算结果。

程序 2 代码如下:

```
#example3.122
s,n=0,0
for i in range(1,11):
```

```
    score=eval(input("输入成绩"))
    if score<60:
        continue
    s=s+score
    n=n+1
print("合格人数为:",n)
print("成绩平均值为:",round(s/n,2))
```

程序运行结果如下：

```
>>>
=======RESTART: C:/ /Python/Python36/example3.122.py=======
输入成绩 87
输入成绩 96
输入成绩 57
输入成绩 82
输入成绩 70
输入成绩 65
输入成绩 60
输入成绩 59
输入成绩 42
输入成绩 90
合格人数为:7
成绩平均值为:78.57
>>>
```

说明：

（1）首先设置初始变量 s 代表总分，n 代表合格人数。

（2）在循环中，如果 if 分支结构判断到输入的 score 小于 60，则忽略下方的求累加和语句 s=s+score 以及记数语句 n=n+1，然后回到 for 语句继续循环下去。

需要注意的是：

（1）在一个双重或者多重循环中，无论是 break 语句还是 continue 语句，都只对当前层次的循环有影响，而对上层循环没有影响。

（2）在带有 else 语句的 for 语句循环和 while 语句循环中，只在循环完整结束时才会执行 else 语句的内容，如果在循环某处执行过 break 语句，则循环被中断，不会执行循环结构中的 else 部分，如果在循环中某处执行过 continue 语句，实际上本次循环依然算被执行过，循环结束后会执行循环结构中的 else 部分。



3.5 结构化程序结构实例

【例 3.13】 自幂数是指一个 n 位数，它的每个位上的数字的 n 次幂之和等于它本身（例如：当 n 为 3 时，有 $1^3+5^3+3^3=153$，153 即是 n 为 3 时的一个自幂数）。当 $n=4$ 时，4 位的自幂数称为四叶玫瑰数，求所有的四叶玫瑰数。

```
#example3.13
for n in range(1000,10000):
    a=int(n/1000)%10
    b=int(n/100)%10
    c=int(n/10)%10
    d=n%10
    if a**4+b**4+c**4+d**4==n:
        print(n,end=";")
```

程序运行结果如下：

```
>>>
=======RESTART: C:/Python/Python36/example3.13.py=======
1634;8208;9474;
>>>
```

说明：

(1) 当输入一个 4 位数字后，首先要求得其数位上的 4 个数字分别是什么，然后再判断各自的 4 次方之和是否等于该数字本身。

(2) 设 n 是任意一个 4 位数，其数位上的数字可以表示如下。

千位 a：a=int(n/1000)%10

百位 b：b=int(n/100)%10

十位 c：c=int(n/10)%10

个位 d：d=n%10

【例 3.14】 输入一个奇数 n，输出 n 行由"*"组成的菱形。例如当输入的 n 为 19 时，输出图形如图 3.7 所示。

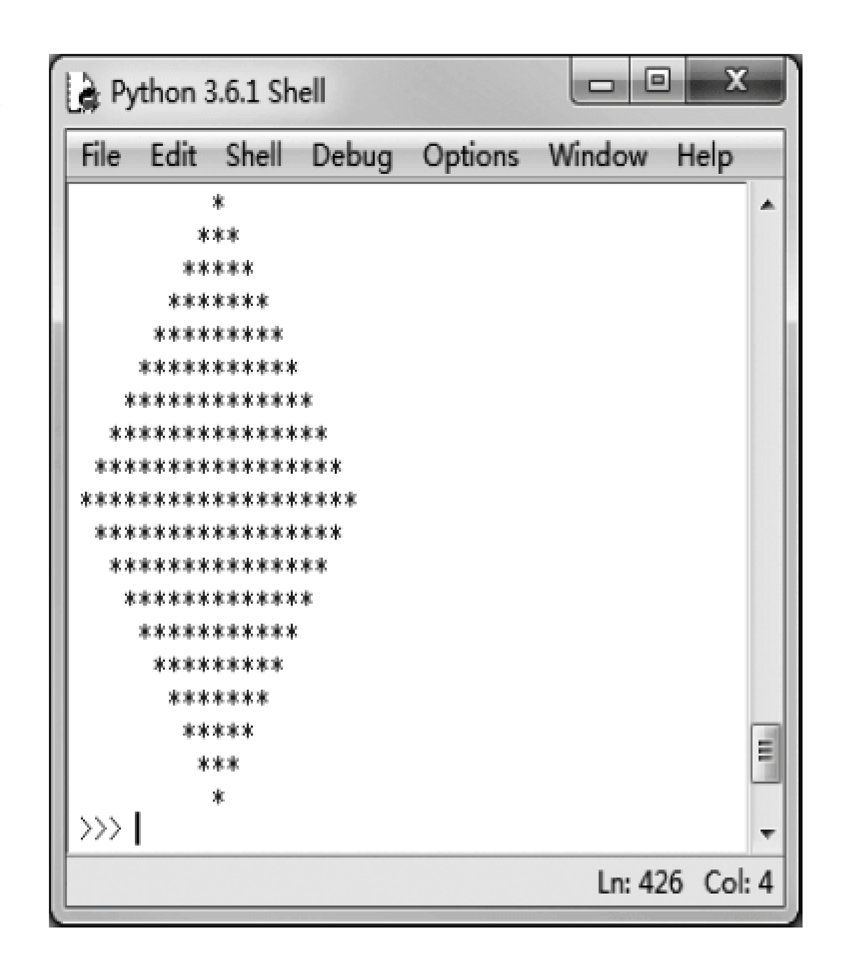


图 3.7 输出星号矩阵

分析：我们把图形分为上下两个三角形进行分析。

对于总体为 n 行的菱形：对于上三角形，总共要输出 $(n+1)/2$ 行；对于下三角形输出 $(n-1)/2$ 行。对于图形中的每一行，都是先输出若干个空格后不换行，再输出若干个星号组成的。先要找到每行空格数、星号数和行号 i 间的关系，归纳成表达式，就可以编写程序了。

在上三角形中，如果最后一行前面没有空格的话，那行号 i 与每行空格数、星号数的对应关系如表 3.1 所示。

在下三角形中，行号与每行空格数，星号数的对应关系如表 3.2 所示。

表 3.1 上三角形对应关系表

行号 i	空格数	星号数
1	$(n+1)/2-1$	1
2	$(n+1)/2-2$	3
3	$(n+1)/2-3$	5
…	…	…
$(n+1)/2$	$(n+1)/2-i$	$2*i-1$

表 3.2 下三角形对应关系表

行号 i	空格数	星号数
1	1	$n-2$
2	2	$n-4$
3	3	$n-6$
…	…	…
$(n-1)/2$	i	$n-2*i$

程序 1 代码如下：

```
#example3.141
from time import sleep
n=eval(input("输入菱形总行数(奇数):"))
up=int((n+1)/2)
down=int((n-1)/2)
for i in range(1,up+1):
print(" "*(up-i),end="")
    print("*"*(2*i-1))
sleep(0.5)
for i in range(1,down+1):
print(" "*i,end="")
    print("*"*(n-2*i))
sleep(0.5)
```

程序运行结果如下：

```
>>>
=======RESTART: C:/ Python/Python36/example3.141.py=======
输入菱形总行数(奇数):7
   *
  * * *
 * * * * *
* * * * * * *
 * * * * *
  * * *
   *
>>>
```

说明：

(1) 从 time 库中导入 sleep 函数，循环中 sleep(0.5)语句的作用是在每输出一行星号后，停顿 0.5s，画面呈现出动态效果。

(2) 输入菱形星号阵的总行数，用 up 和 down 两个变量表示上三角形的行数和下三角形的行数，用两个 for 语句循环分别画出上三角形和下三角形中的每一行，由于菱形的上三角形总比下三角形多一行，所以变量 down 也可以省略，而用 up-1 代替。

如果在循环结构中用一个 if 分支结构判断画的是上三角形还是下三角形，就可以用一个循环结构画出菱形，如以下程序：

```
#example3.142
from time import sleep
n=eval(input("输入菱形总行数(奇数):"))
up=int((n+1)/2)
for i in range(1,n+1):
    if i<=(n+1)/2:
    print(" "*(up-i),end="")
    print("*"*(2*i-1))
    else:
    print(" "*(i-up),end="")
    print("*"*(n-2*(i-up)))
sleep(0.5)
```

程序运行结果如下：

```
>>>
=======RESTART: C:/Python/Python36/example3.142.py=======
```

```
输入菱形总行数(奇数):9
    *
   * * *
  * * * * *
 * * * * * * *
* * * * * * * * *
 * * * * * * *
  * * * * *
   * * *
    *
>>>
```

说明:

(1) 程序用 up 代表上三角形的行数。

(2) if 语句分支中,当 i<=(n+1)/2 时,画的是上三角形,否则画下三角形,由于只用了一个循环,所以 i 为菱形中的当前行号,画下三角形时,下三角形的行号应该为 i-up,即当前画的行数减去上三角形占用的行数。



习题 3

一、填空题

1. 在程序流程图中,菱形一般代表着________。

2. Python 通过________符号判断是否存在分支结构。

3. 在多分支结构中,除了第一个判断需要使用 if 关键字,其他的判断要使用________关键字。

4. Python 语言的循环结构可以分为 for 语句循环和________语句循环。

5. 语句 print(3<=4<5)的值为________。

6. 可以结束一个循环的关键字是________。

二、程序设计

1. 输入三角形的三条边长,如果输入的边长满足三角形的三边关系,则应用海伦公式 $s=\sqrt{p*(p-a)*(p-b)*(p-c)}$ 计算三角形的面积,其中:a、b、c 代表三角形的三条边长,s 代表面积,p 为三角形周长的一半。当输入的边长不构成三角形时,提示用户输入

错误。

2. 编程求一元二次方程 $y=ax^2+bx+c$ 的根。

3. 输入一个年份,如果该年份能被 400 整除,则为闰年;如果年份能被 4 整除但不能被 100 整除也为闰年。编程判断输入的年份是否为闰年。

4. 输入两个数字,求两个数字的最大公约数。

5. 输入一个数字,判断该数字是否为素数(只能被 1 和自身整除的数字称为素数)。

6. 输入一个字符串,将该字符串逆序输出。

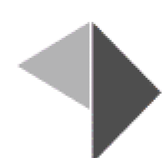
7. 2011 年,我国人口为 13.47 亿,人口增长率为 0.48%,同年印度总人口为 12.1 亿,人口增长率为 1.2%。假设人口增长率不变,求印度的人口将在哪年超过中国。

第 4 章

Python 数据结构

学习目标

- 掌握列表的创建、访问和常见的更新操作
- 掌握元组的创建和访问
- 掌握字典的创建、访问和常见的更新操作
- 掌握集合的创建、访问和常见的更新操作
- 了解如何利用推导式创建列表、元组和字典



4.1 组合类型简介

Python 可以处理的数据类型已经在第 2 章介绍过，如整型、字符串类型和逻辑型等，这些数据类型只能表示一个单一的数据，称为基本数据类型。但是在处理多个有关联的数据时，仅仅使用基本数据类型是不够的。除了这些简单数据类型外，Python 语言还可以处理一些复杂的数据类型，称为组合数据类型。

组合数据类型能够将多个基本数据类型或组合数据类型组织起来，作用是能够更清晰地反映数据之间的关系，也能更加方便地管理和操作数据。Python 中组合数据类型有三类：序列类型、映射类型和集合类型。

序列类型是一个元素向量，元素之间存在先后关系，通过序号访问。Python 中的序列类型主要有字符串(str)，列表类型(list)和元组类型(tuple)等。

映射类型是一种键值对，一个键只能对应一个值，但是多个键可以对应相同的值，通过键可以访问值。字典类型(dict)是 Python 中唯一的映射类型。字典中的元素没有特定的顺序，每个值都对应一个唯一的键。字典类型的数据和序列类型的数据的区别在于存储和访问方式的不同。另外，序列类型只用整数作为序号，而映射类型可以用整数、字符串或者其他类型的数据作为键，而且键和键值有一定关联性，也就是键可以映射到数值。

集合类型是通过数学中的集合概念引进的，是一种无序不重复的元素集。集合的元素类型只能是固定数据类型，例如整型、字符串、元组等，而列表、字典等是可变数据类型，不能作为集合中的数据元素。集合可以进行交、并、差、补等运算，含义与数学中的相应概念相同，如图 4.1 所示。

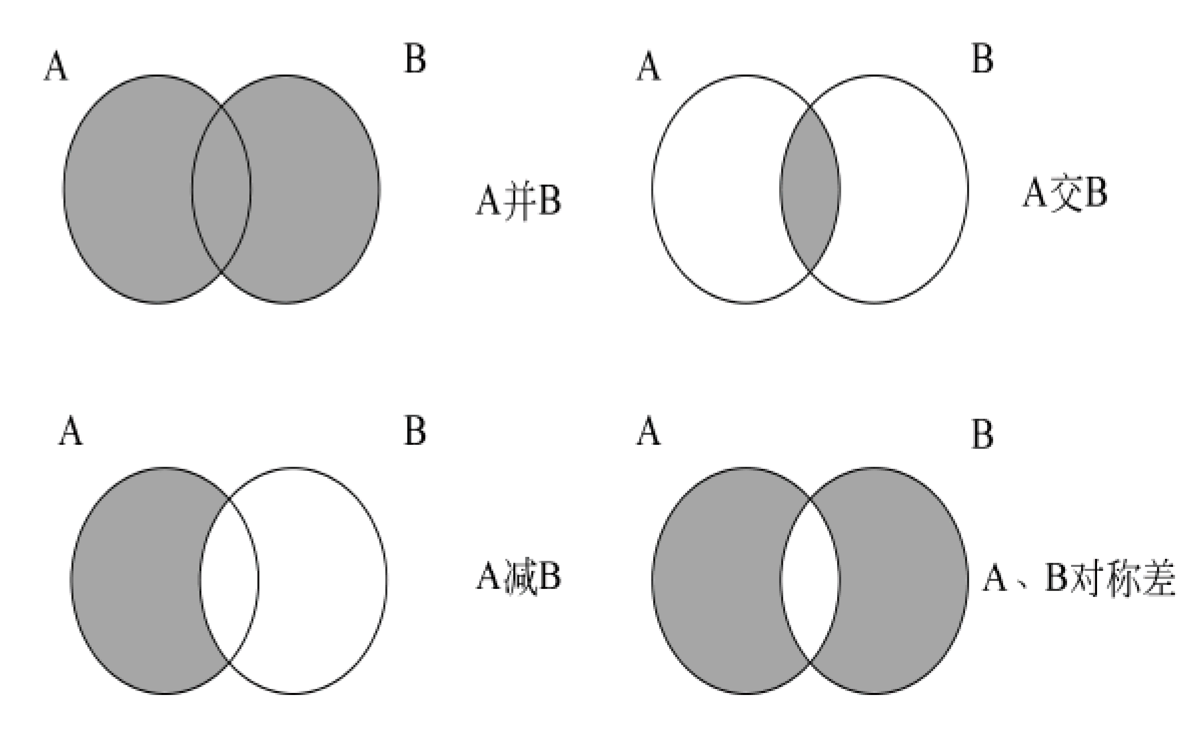


图 4.1　集合的四种基本运算



4.2　列表

列表是一种可变序列数据类型。列表将数据元素放在一对方括号之间，并使用逗号作为数据元素的分割。一个列表中的数据元素可以是基本数据类型，也可以是组合数据类型或自定义数据类型的数据。例如下面的列表都是合法的列表对象：

```
[1,2,3,4,5,6,7]
[1, "10086","中国移动",True]
["日期","中国",[2017,5,1]]
[1,[2,3,4],(5,6),{"a":7,"b":8,"c":9}]
```

4.2.1　创建列表

创建一个列表，可以通过使用方括号，并且把方括号内每一个列表元素用逗号分开进行创建，或者使用 list()函数进行创建。可以采用以下方法创建列表：

```
>>>list1=[]                              #产生一个空列表
>>>list1
[]
```

```
>>>list2=[1,2,3,4,5,6]
>>>list2
[1, 2, 3, 4, 5, 6]
>>>list3=['Python','c++','Java','VB','Perl']
>>>list3
['Python', 'c++', 'Java', 'VB', 'Perl']
>>>list4=list()                    #产生一个空列表,等价于 list1=[]
>>>list4
[]
>>>list5=list(range(1,10,2))       #将 range 函数产生的序列变为列表
>>>list5
[1, 3, 5, 7, 9]
>>>list6=list("沈阳师范大学")       #每个字符作为列表中的一个数据元素
>>>list6
['沈', '阳', '师', '范', '大', '学']
>>>
```

除了手动添加列表中的数据外,也可以使用列表推导式创建。列表推导式可以使用非常简洁的方式生成满足特定需要的列表。语法格式如下:

```
[<表达式>for<变量>in<序列>]
```

【例 4.1】 使用列表推导式创建列表。

```
#example4.1
>>>list1=[x*x for x in range(1,10)]
>>>list1
[1, 4, 9, 16, 25, 36, 49, 64, 81]
>>>list2=[i for i in list1 if i%2==0]
>>>list2
[4, 16, 36, 64]
>>>list3=[i for i in range(100,1000) if (int(i/100))**3+(int(i/10)%10)**3+(i%10)
**3==i]
>>>list3
[153, 370, 371, 407]
>>>
```

说明：

(1) list1 是由 10 以内的自然数的平方组成的列表。

(2) list2 是由 list1 中的偶数组成的列表。

(3) list3 是由水仙花数组成的列表，水仙花数即 3 位自幂数，请参考例 3.13。

4.2.2　访问列表

列表是一个有序序列，可以通过序号访问列表中的列表元素。和字符串类似，列表的序号也是一个整数，并且可以由正向或逆向访问列表，如图 4.2 所示。

正向索引，从 0 开始递增					
0	1	2	3	4	5
['沈',	'阳',	'师',	'范',	'大',	'学']
−6	−5	−4	−3	−2	−1
逆向索引，从-1 开始递减					

图 4.2　列表的索引序号

通过使用索引号，可以访问到列表中的某些列表元素，格式如下：

```
<列表名>[<索引>]
```

其中，列表名为一个列表的名字，索引为访问的列表元素序号。

例如：

```
>>>list1=['a','b','c','d','e','f','g']
>>>list1[0]          #访问列表 list1 中正向索引序号为 0 的列表元素
'a'
>>>list1[2:4]        #访问列表 list1 中正向索引序号从 2 到 4(不包括)的列表元素
['c', 'd']
>>>list1[-2]         #访问列表 list1 中逆向索引序号为-2 的列表元素
'f'
>>>list1[4:]         #访问列表 list1 中正向索引序号从 4 到最后的列表元素
['e', 'f', 'g']
>>>list1[:-4]        #访问列表 list1 中从开始到逆向索引序号为-4(不包含)的列表元素
['a', 'b', 'c']
>>>list1[8]          #如果访问的列表序号不存在，则返回索引序号错
…
```

```
IndexError: list index out of range        #如果访问的列表序号不存在,则返回索引序号错
>>>
```

如果一个列表中的列表元素也是由列表构成的,那就构成了类似矩阵的二维结构。访问二维列表的格式如下:

```
<列表名>[<索引 1>][<索引 2>]
```

其中,索引 1 为二维列表的元素索引号,索引 2 为二维列表中索引 1 指向的列表元素中的元素索引号。

例如:

```
>>>a=[1,2,3,4,5]
>>>b=[6,7,8,9,10]
>>>c=[11,12,13,14,15]
>>>list=[a,b,c]
>>>list[1][3]
9
>>>
```

语句 list[1][3]中,1 代表访问的是 list 列表的索引序号 1 的列表元素,即列表 b;3 代表访问列表 b 中索引序号为 3 的列表元素,即数字 9。

【例 4.2】 遍历二维列表。

```
#example4.2
list1=['20170001','赵明','男',19,'物理学院','流体力学']
list2=['20170002','钱小红','女',20,'化学学院']
list3=['20170003','孙强','男',20,'信息学院','计算机科学与技术','3班']
list4=['20170004','李丽','女',19,'外语学院']
list=[list1,list2,list3,list4]
for i in range(len(list)):
    for j in range(len(list[i])):
        print(list[i][j],end=" ")        #end=""负责输出每项数据后加个空格
    print()
```

程序运行结果如下:

```
>>>
===============RESTART: C: /Python/Python36/example4.2.py============
```

```
20170001 赵明男 19 物理学院流体力学
20170002 钱小红女 20 化学学院
20170003 孙强男 20 信息学院计算机科学与技术 3班
20170004 李丽女 19 外语学院
>>>
```

说明：

(1) 列表 list1 到 list4 为 4 名同学的自然情况，列表 list 是包含了以上 4 个列表的列表。

(2) len(list)用来求列表 list 的列表元素个数，range(len(list))可以返回 list 中列表元素的正向索引序号。

(3) 在双重循环中，外层循环遍历列表 list 中的每个列表元素，内层循环遍历 list[i]中的每个列表元素。

(4) print()作用为输出一行数据后换行。

列表也可以像变量之间的赋值那样，将一个列表的值赋给另一个列表，但是和基本变量赋值不同的是，列表的赋值只是将实际数据的地址引用进行了赋值，而不是将实际数据赋值一份给新的列表。如：

```
>>>var1=100
>>>var2=var1
>>>var1,var2
(100, 100)                          #两个变量的值相等
>>>var2=200
>>>var1,var2
(100, 200)                          #更改一个变量的值,另一个不会变
>>>list1=[1,2,3,4]
>>>list2=list1                      #将 list1 赋值给 list2
>>>list2[0]=5                       #更改 list2 中的某些元素值
>>>list2[1]=6
>>>list1,list2
([5, 6, 3, 4], [5, 6, 3, 4])  #list1 和 list2 同时改变。可以看出,list2 使用了 list1 的存储
                                地址。当改变 list2 的元素,list1 的元素同时改变
>>>list2=[10,11,12,13]        #对 list2 进行实际数据的赋值,list2 会使用独立的存储地址,不
                                再和 list1 关联
>>>list1,list2
([5, 6, 3, 4], [10, 11, 12, 13])
>>>
```

4.2.3 更新列表

列表是一种可变的数据类型，列表的长度和列表元素的值都是可以更改的。更新列表主要有修改列表元素、添加列表元素和删除列表元素等操作。

1. 修改列表元素值

修改列表元素值可以用赋值语句，语法格式如下：

```
<列表名>[<索引>]=<值>
```

其中，列表名为一个已经存在的列表，索引为该列表的正向或逆向索引序号，值为任意数据值。当索引不在列表的索引范围内时，会提示用户“索引超出范围”错误。例如：

```
>>>#修改列表中的一个列表元素
>>>fruit=['苹果','香蕉','西瓜','桔子','桃子']
>>>fruit[0]='apple'
>>>fruit[-1]='peach'
>>>fruit
['apple', '香蕉', '西瓜', '桔子', 'peach']
>>>#修改列表中的多个元素(也具有增加,删除功能)
>>>fruit[1:4]=['banana','watermelon','orange']          #当 Index 为一个范围时,值也需是列表
>>>fruit
['apple', 'banana', 'watermelon', 'orange', 'peach']
>>>#当列表序号范围和赋值列表长度不相等时,就可以增加或删除列表
>>>fruit[0:2]=['苹果','香蕉','梨','樱桃']#用 4 个列表元素替换选定的两个列表元素
>>>fruit
['苹果', '香蕉', '梨', '樱桃', 'watermelon', 'orange', 'peach']
>>>fruit[-3:]=['柠檬']                    #用一个列表元素替换选定的 3 个列表元素
>>>fruit
['苹果', '香蕉', '梨', '樱桃', '柠檬']
>>>
```

2. 添加列表元素

虽然在对列表赋值时可以添加数据元素，但通常还是使用专门的函数对列表进行添加元素操作。append()函数可以向列表的最后添加列表元素；insert()函数可以向列表中

的指定索引序号处插入列表元素。语法格式如下：

```
<列表名>.append(<值>)
<列表名>.insert(<索引>,<值>)
```

例如：

```
>>>fruit=['苹果','香蕉','西瓜','桔子','桃子']
>>>fruit.append("樱桃")
>>>fruit
['苹果', '香蕉', '西瓜', '桔子', '桃子', '樱桃']
>>>fruit.insert(2,"柠檬")
>>>fruit
['苹果', '香蕉', '柠檬', '西瓜', '桔子', '桃子', '樱桃']
>>>
```

注意，这两个函数每次只能插入一个列表元素。

3. 删除列表元素

删除列表元素通常可以使用 remove()函数或者 del 语句进行操作。remove()函数用来按值删除列表中的列表元素，del 语句可以按索引序号删除列表中的列表元素。语法格式如下：

```
<列表名>.remove(<值>)
del<列表名>[<索引>]或者 del<列表名>
```

其中，列表名为一个已经存在的列表名称，值为在列表中的列表元素，索引为列表索引序号。remove()函数可以删除列表中第一个和数据值相等的列表元素，如果列表中有 1 个以上相同的列表元素，要多次使用 remove()函数。del 语句可以删除列表中的列表元素，也可以删除整个列表。例如：

```
>>>fruit=['苹果','香蕉','西瓜','桔子','西瓜','桃子']
>>>fruit.remove('西瓜')
>>>fruit
['苹果', '香蕉', '桔子', '西瓜', '桃子']          #remove 函数使用一次只能删除一个"西瓜"
>>>del fruit[:2]                          #del 语句删除列表中 0,1 索引序号的列表元素
>>>fruit
['桔子', '西瓜', '桃子']
```

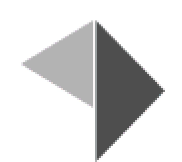
```
>>>del fruit              #删除列表
>>>fruit                  #列表删除后,再使用列表,将出现"未定义"错误
...
NameError: name 'fruit' is not defined
>>>
```

4.2.4 列表常用的其他操作

除了创建列表、删除列表、在列表中进行增删改查等操作以外,列表还支持很多其他操作。表 4.1 列出了常用的列表操作符和函数。

表 4.1 常用列表操作

操　作	功　能
list1+list2	连接两个列表,新列表的列表元素的个数为 list1 和 list2 列表元素个数之和
list1 * n 或 n * list1	将 list1 重复 n 次
x in list1	如果列表 list1 包含 x,则返回 True,否则返回 False
x not in list1	如果列表 list1 中不包含 x 对象则返回 True,否则返回 False
cmp(list1,list2)	比较两个列表中的元素,如果 list1 中的元素比 list2 中对应的元素大,则函数返回 1;如果小,则函数返回-1;如果 list1 和 list2 中所有元素相等,则函数返回 0
len(list1)	函数返回 list1 中列表元素的个数
max(list1)	函数返回 list1 列表中元素的最大值,要求 list1 中列表元素类型相同
min(list1)	函数返回 list1 列表中元素的最小值,要求 list1 中列表元素类型相同
sorted(list1)	函数返回 1 个新列表,将 list1 中的元素按升序排序
reversed(list1)	函数返回 1 个新列表,新列表将 list1 翻转,即第 1 个元素与最后一个对换;第 2 个与倒数第 2 个对换,依此类推
sum(list1)	如果 list1 中所有列表元素都是数字,则函数返回列表元素之和



4.3 元组

元组和列表类似,但元组的元素是不可变的,元组一旦创建,不可以修改其元素,也不能添加或者删除元素。创建一个元组可以使用一对小括号,在小括号内用逗号分隔元组元素。一个元组中的数据元素可以是基本数据类型,也可以是组合数据类型或自定义数

据类型的数据。例如,下面的元组都是合法的:

```
(1,2,3,4,5)
("Python","C#","Java","Go","VB")
()
(5,)
((1,2,3),("a","b","c"),(True,False))
```

需要注意的是,只含有一个元素的元组,元素后面一定要有逗号,否则就是一个表达式,而不是元组了。两个或者两个以上元素的元组,最后一个元素后可以有逗号,也可以没有。

4.3.1　创建元组

创建一个元组,可以通过使用小括号,并且把小括号内每一个元组元素用逗号分开进行创建,或者使用 tuple()函数进行创建。

```
>>>tup1=()                              #产生一个空元组
>>>tup1
()
>>>tup2=(1,2,3,4,5,)
>>>tup2
(1, 2, 3, 4, 5)
>>>tup3=('Python','c++','Java','VB','Perl')
>>>tup3
('Python', 'c++', 'Java', 'VB', 'Perl')
>>>tup4=tuple()                         #产生一个空元组,等价于 tup1=()
>>>tup4
()
>>>tup5=tuple(range(1,10,2))            #将 range 函数产生的序列变为元组
>>>tup5
(1, 3, 5, 7, 9)
>>>tup6=tuple("沈阳师范大学")            #每字符作为元组中的一个数据元素
>>>tup6
('沈', '阳', '师', '范', '大', '学')
>>>
```

使用小括号创建元组时,小括号也可以省略,例如:

```
>>>tup=1,2,3,4,5
>>>tup
(1, 2, 3, 4, 5)
>>>
```

和列表一样,元组一样也可以使用推导式生成,语法也近似,只是不用中括号而用小括号:

```
(<表达式>for<变量>in<序列>)
```

但是和列表推导式不同的是,这种推导式称为生成器推导式,它的结果是一个生成器对象,而不是元组。生成器对象可以使用 next()函数依次访问其中的元素,也可以使用 list()函数或者 tuple()函数转化为列表或者元组后使用。例如:

```
>>>g=(x**2 for x in range(1,10))            #利用推导式产生生成器 g
>>>g
<generator object<genexpr>at 0x02D03900>    #访问 g 会提示在某地址有生成器
>>>next(g)                       #使用 next 函数访问生成器中第一个元素
1
>>>next(g)                       #使用 next 函数继续向下访问生成器中的元素
4
>>>next(g)
9
>>>tuple1=tuple(g)               #使用 tuple()函数将生成器中没访问的元素生成为元组
>>>tuple1
(16, 25, 36, 49, 64, 81)
>>>
```

列表推导式产生的列表与生成器推导式产生的生成器有本质区别。

(1) 列表推导式会直接形成一个新的列表,一次性把列表中的所有数据都放入内存中,如果列表元素非常多,会占用很大的内存空间,生成器只是生成器推导式指定的算法,当访问生成器时才产生具体的元素,而不是一次性生成所有元素,所以生成器只占用很小的内存空间。

(2) 列表推导式生成的列表中的元素可以多次访问,生成器推导产生的生成器中的元素只能从前向后一个元素一个元素地访问,访问后即消失,例如上述例子中,当访问了生成器中的元素 1、4、9 以后,再将生成器 g 转化为元组,只能转化生成器中未访问的元素,即把 16 到 81 转化为元组,生成器使用过一次以后就被释放,如需再次使用,只能再用

生成器推导式重新产生，如：

```
>>>g=(x**2 for x in range(1,10))
>>>for i in g:
            print(i,end=",")
1,4,9,16,25,36,49,64,81,
>>>for i in g:
            print(i,end=",")
>>>
```

注意：第一个 for 语句循环遍历生成器 g 中的所有元素后，第二个 for 语句循环就没有任何结果了，因为生成器中的所有元素都已经访问完，不能回到生成器的第一个元素再次访问了，所以第二个循环没有任何显示。

当需要重复访问一个生成器中的元素时，可以根据需要用 list()函数转为列表或使用 tuple()函数转为元组后使用。

4.3.2　访问元组

元组属于不可变序列，一旦创建，元组中的值就固定不可变了，也无法为元组增加元素或者删除元素。可以通过切片访问为列表增加或者删除元素，但是元组不可以。访问一个元组和访问列表的方法类似。例如：

```
>>>tuple1=('a','b','c','d','e','f','g')
>>>tuple1[1]          #访问元组 tuple1 中正向索引序号为 1 的元素
'b'
>>>tuple1[3:5]        #访问元组 tuple1 中正向索引序号从 3 到 4 的元素
('d', 'e')
>>>tuple1[-1]         #访问元组 tuple 中逆向索引序号为-1 的元组
'g'
>>>
```

如果一个元组中的元素也是由列表和元组等构成的，也会构成二维结构。可以采用和访问与二维列表相似的方式访问二维元组。虽然元组不可改变，但是如果元组元素是列表等可变序列，该列表是可以改变的。例如：

```
>>>a=[1,2,3]
>>>b=[4,5,6]
>>>c=(7,8,9)
```

```
>>>d=(a,b,c)                    #元组d由2个列表和1个元组组成
>>>d
([1, 2, 3], [4, 5, 6], (7, 8, 9))
>>>d[1]=[10,20,30]              #修改元组元素,会产生错误,因为元组是不可变的
Traceback (most recent call last):
File "<pyshell#113>", line 1, in<module>
d[1]=[10,20,30]
TypeError: 'tuple' object does not support item assignment
>>>d[1][0]=10                   #修改1号元组元素中0号列表元素
>>>d[1][1]=20                   #修改1号元组元素中1号列表元素
>>>d[1][2]=30                   #修改1号元组元素中2号列表元素
>>>d                            #修改成功
([1, 2, 3], [10, 20, 30], (7, 8, 9))
>>>d[2][0]=70                   #修改2号元组元素中0号元组元组,错误
Traceback (most recent call last):
File "<pyshell#118>", line 1, in<module>
d[2][0]=70
TypeError: 'tuple' object does not support item assignment
>>>
```

元组常用操作也是+和*,元组常用函数也和列表常用函数基本一致,请参考列表操作,本节不再叙述。

4.4 字典

字典是一种映射型的组合数据结构,由键值对组成。在一个字典结构中,一个键只能对应一个值,但是多个键可以对应相同的值。这就好像在一个学校里,一个学生只能有唯一的学号,但是具有多个不同学号的学生姓名有可能相同。字典类型和序列类型的区别在于存储和访问方式都不同。序列类型通常通过索引序号访问,而且只采用整数作为索引序号;字典可以用任意不可变类型(如数字、字符串、元组等)作为值的索引序号,也就是键。

字典将键值对放在一对大括号之间,并使用逗号作为分隔,每个键值对内部用冒号分隔。例如下面的字典都是合法的:

```
{}
{1:"Python",2:"C++",3:"Java",4:"VB"}
{"No1":"Python","No2":"C++","No3":"Java","No4":"VB"}
```

```
{("三班",15):["张晓红","女"],("四班",22):["董强","男"],("五班",7):["张晓红","女"]
```

4.4.1　字典的创建

字典对象是由键值对组成的，外面是大括号。可以采用以下方法创建字典。

1. 通过赋值语句创建字典

```
>>>dict1={}
>>>dict1
{}
>>>dict2={'优':90,'良':80,'中':70,'及':60}
>>>dict2
{'优': 90, '良': 80, '中': 70, '及': 60}
```

2. 通过 dict()函数创建字典

使用 dict()函数可以将键值对形式的列表创建为字典。例如：

```
>>>dict3=dict([['优', 90], ['良', 80], ['中', 70], ['及', 60]])
>>>dict3
{'优': 90, '良': 80, '中': 70, '及': 60}
```

使用 dict()函数也可以将键值对形式的元组创建为字典。

```
>>>dict4=dict((('优', 90), ('良', 80), ('中', 70), ('及', 60)))
>>>dict4
```

在 dict()函数中调用 zip()函数，可以将多个序列作为参数，返回由元组构成的列表。下面例子中，将两个列表['优','良','中','及']和[90,80,70,60]作为 zip()函数的参数，则函数结果为[('优',90),('良',80),('中',70),('及',60)]，再通过 dict()函数即可创建字典：{'优':90,'良':80,'中':70,'及':60}。

```
>>>dict5=dict(zip(['优', '良', '中', '及'],[90,80,70,60]))
>>>dict5
{'优': 90, '良': 80, '中': 70, '及': 60}
```

3. 通过 fromkeys()函数创建字典

调用 fromkeys()函数可以创建值都相同的字典，可以将一个序列作为键，然后指定一个统一的值。formkeys()函数也可以不指定值，创建的字典默认为 None 空值。例如：

```
>>>dict6={}.fromkeys(['优', '良', '中', '及'],"大于 60分")
>>>dict6
{'优': '大于 60分', '良': '大于 60分', '中': '大于 60分', '及': '大于 60分'}
>>>dict7={}.fromkeys(['优', '良', '中', '及'])
>>>dict7
{'优': None, '良': None, '中': None, '及': None}
>>>
```

4. 通过推导式创建字典

也可以使用推导式创建字典，语法格式如下：

```
{<键>:<值>for<变量>in<序列>}
```

例如：

```
>>>dict8={n: n* * 2 for n in range(1,5)}
>>>dict8
{1: 1, 2: 4, 3: 9, 4: 16}
>>>
```

需要注意的是，字典是一个无序序列，在内部存储时，不一定与创建字典的键值对顺序相同，所以在访问字典时，不需要特意关注显示的前后顺序。

4.4.2 访问字典

1. 通过键访问值

因为字典的键值对是一种映射关系，根据键就可以访问到值。所以访问字典中键值对的值可以通过方括号并指定键的方式进行访问。如果键不存在，则访问错误。例如：

```
>>>dict4                                #设已有字典 dict4,内容如下
{'优': 90, '良': 80, '中': 70, '及': 60}
>>>dict4['良']                          #访问键为"良"的映射值
80
>>>dict4['良好']                        #访问键为"良好"的映射值,字典中没有该键,会出错
Traceback (most recent call last):
    File "<pyshell#31>", line 1, in<module>
    dict4['良好']
KeyError: '良好'
>>>
```

字典对象也提供了一个 get()方法通过键访问对应的值：当键存在时，返回该键对应的值，而键不存在时也不会出错，而返回指定值。例如：

```
>>>dict4
{'优': 90, '良': 80, '中': 70, '及': 60}
>>>dict4.get('良','键不存在')
80
>>>dict4.get('良好','键不存在')
键不存在'
>>>
```

2. 访问字典的所有键值对、所有键、所有值

通过调用字典的 items()方法可以返回所有的键值对；调用 keys()方法可以返回所有的键；而调用 value()方法可以返回所有的值。例如：

```
>>>dict4
{'优': 90, '良': 80, '中': 70, '及': 60}
>>>dict4.items()
dict_items([('优', 90), ('良', 80), ('中', 70), ('及', 60)])
>>>dict4.keys()
dict_keys(['优', '良', '中', '及'])
>>>dict4.values()
dict_values([90, 80, 70, 60])
>>>
```

3. 遍历字典

一般通过 for 语句循环可以遍历字典的元素。例如：

```
  >>>dict4
{'优': 90, '良': 80, '中': 70, '及': 60}
>>>for i in dict4:                #默认访问的是字典的键
  print(i,end=",")
优,良,中,及,
>>>for i in dict4.items():        #指定访问字典键值对元素
  print(i,end=",")

('优', 90),('良', 80),('中', 70),('及', 60),
>>>for i in dict4.values():       #指定访问字典的值
  print(i,end=",")

90,80,70,60,
>>>
```

4.4.3 更新字典

字典和列表一样，是一个可变对象，所以对字典可以进行增加元素、删除元素、修改元素等操作。

1. 添加元素

通过赋值语句可以向字典添加键值对元素。例如：

```
>>>dict1={'No1': 'Python', 'No2': 'C++', 'No3': 'Java', 'No4': 'VB'}
>>>dict1
{'No1': 'Python', 'No2': 'C++', 'No3': 'Java', 'No4': 'VB'}
>>>dict1['No5']='PHP'                  #若键不是字典原有键,则添加一个键值对元素
>>>dict1
{'No1': 'Python', 'No2': 'C++', 'No3': 'Java', 'No4': 'VB', 'No5': 'PHP'}
>>>dict1['No2']='C#'                 #若键是字典原有键,则更新该键的值,不添加元素
>>>dict1
{'No1': 'Python', 'No2': 'C#', 'No3': 'Java', 'No4': 'VB', 'No5': 'PHP'}
```

使用 setdefault()函数向字典添加元素。例如：

```
>>>dict1={'No1': 'Python', 'No2': 'C++', 'No3': 'Java', 'No4': 'VB'}
>>>dict1
{'No1': 'Python', 'No2': 'C++', 'No3': 'Java', 'No4': 'VB'}
>>>dict1.setdefault('No5','PHP')
                        #若第一个参数不是字典原有键,则把第二参数作为该键的值构成
                         一个键值对添加到字典中,并返回函数结果就是该键值对的值
'PHP'
>>>dict1
{'No1': 'Python', 'No2': 'C++', 'No3': 'Java', 'No4': 'VB', 'No5': 'PHP'}
>>>dict1.setdefault('No2')
                          #若第一个参数是字典的原有键,则返回该键的值。原字典不变
'C++'
>>>dict1
{'No1': 'Python', 'No2': 'C++', 'No3': 'Java', 'No4': 'VB', 'No5': 'PHP'}
>>>dict1.setdefault('no2')
          #若第一参数不是字典原有键,又省略了第二参数,则向字典中添加"该键:None"元素
>>>dict1
{'No1': 'Python', 'No2': 'C++', 'No3': 'Java', 'No4': 'VB', 'No5': 'PHP', 'no2':None}
>>>
```

2. 合并字典

使用 update()函数可以将一个字典中的元素添加到当前字典中,如果两个字典的键有重名,则用另一个字典中的值对当前字典进行更新。

```
>>>dict1={'No1': 'Python', 'No2': 'C++'}
>>>dict2={'No3': 'Java', 'No4': 'VB', 'No5': 'PHP'}
>>>dict3={'No2': 'C#'}
>>>dict1.update(dict2)
                #若 dict2 和 dict1 的键没有重复,则将 dict2 中的所有元素添加到 dict1 中
>>>dict1
{'No1': 'Python', 'No2': 'C++', 'No3': 'Java', 'No4': 'VB', 'No5': 'PHP'}
>>>dict1.update(dict3)          #若 dict3 中有和 dict1 重复的键,则用 dict2 中重复键"
                                 No2"的值"c#"更新 dict1 中"No2"键的值
>>>dict1
{'No1': 'Python', 'No2': 'C#', 'No3': 'Java', 'No4': 'VB', 'No5': 'PHP'}
>>>
```

3. 修改元素的值

在添加元素中已经说过,修改字典中的元素的值可以用赋值语句实现。例如:

```
>>>dict1={'No1': 'Python', 'No2': 'C++'}
>>>dict1
{'No1': 'Python', 'No2': 'C++'}
>>>dict1['No2']='Java'
>>>dict1
{'No1': 'Python', 'No2': 'Java'}
>>>
```

4. 删除元素

删除字典中的元素可以用 del()函数、del 语句、pop()函数、popitem()函数或者 clear()函数实现。例如:

```
>>>dict1={'No1': 'Python', 'No2': 'C++', 'No3': 'Java', 'No4': 'VB'}
>>>del(dict1['No4'])            #使用 del 函数删除指定键的字典元素
>>>dict1
{'No1': 'Python', 'No2': 'C++', 'No3': 'Java'}
>>>del dict1['No3']             #使用 del 语句删除指定键的字典元素
>>>dict1
{'No1': 'Python', 'No2': 'C++'}
>>>del dict1                    #使用 del 语句删除整个字典
>>>dict1                        #若字典删除后再进行访问,则返回错误信息
Traceback (most recent call last):
File "<pyshell#100>", line 1, in<module>
    dict1
NameError: name 'dict1' is not defined
>>>dict1={'No1': 'Python', 'No2': 'C++', 'No3': 'Java', 'No4': 'VB'}
>>>dict1.pop('No4','键不存在')     #使用 pop 函数删除指定键的字典元素,如果该键不存在
则返回第二参数的值,如果键存在,则返回该键的值,同时删除
'VB'
>>>dict1
{'No1': 'Python', 'No2': 'C++', 'No3': 'Java'}
>>>dict1.pop('No5','键不存在')
'键不存在'
>>>dict1={'No1': 'Python', 'No2': 'C++', 'No3': 'Java', 'No4': 'VB'}
```

```
>>>dict1.popitem()                #使用 popitem()函数可以随机删除字典中一个元素,并
返回该元素。因为字典里的元素是无序的,所以相当于在集合中去掉一个元素,并没有顺序。当
字典为空时,再使用 popitem()函数会返回错误
('No4', 'VB')
>>>dict1.popitem()
('No3', 'Java')
>>>dict1.popitem()
('No2', 'C++')
>>>dict1.popitem()
('No1', 'Python')
>>>dict1.popitem()
Traceback (most recent call last):
File "<pyshell#122>", line 1, in<module>
    dict1.popitem()
KeyError: 'popitem(): dictionary is empty'
>>>dict1={'No1': 'Python', 'No2': 'C++', 'No3': 'Java', 'No4': 'VB'}
>>>dict1.clear()                  #使用 clear()函数可以清空字典,但字典依然存在
>>>dict1
{}
>>>
```

4.4.4　字典常用的其他操作

除了创建字典、访问字典、更新字典外,字典还有很多其他操作,表 4.2 列出了字典的常用操作符和函数。

表 4.2　常用字典操作

操　作	功　能
key in dict1	如果字典 dict1 包含键 key,则返回 True,否则返回 False
(key,value) in dict1.items()	如果字典 dict1 包含键值对(key,value),则返回 True,否则返回 False
value in dict1.values()	如果字典 dict1 包含值 value,则返回 True,否则返回 False
len(dict1)	求字典 dict1 中的元素个数
max(dict1)	求字典 dict1 键的最大值,要求所有键数据类型相同
min(dict1)	求字典 dict1 键的最小值,要求所有键数据类型相同

续表

操　作	功　能
list(dict1)	返回由字典 dict1 中的键组成的列表
list(dict1.items())	返回由字典 dict1 的元素组成的列表,键值对转化为元组(key,value)
list(dict1.values())	返回由字典 dict1 的值组成的列表

【例 4.3】 随机产生 500 个小写字母,统计每个字母出现的频率。

```
#example4.3
from random import choice
str='abcdefghijklmnopqrstuvwxyz'
dict1=dict()
for i in range(500):
    r=choice(str)
    dict1[r]=dict1.get(r,0)+1
print('一共产生的字母为:',sum(dict1.values()))
print('每个字母产生的词频为:',dict1)
```

程序运行结果如下(结果为随机产生,每次运行不同):

```
>>>
==============RESTART: C:/ Python/Python36/example4.3.py=============
一共产生的字母为: 500
每个字母产生的词频为: {'y': 23, 'c': 18, 'e': 21, 'h': 17, 'v': 22, 'u': 25, 'r': 23, 'q':
23, 'x': 19, 'b': 17, 'a': 23, 'j': 25, 'o': 23, 'g': 14, 'n': 17, 'f': 24, 'i': 17, 'p': 26, '
t': 21, 'm': 18, 's': 17, 'w': 14, 'k': 14, 'l': 17, 'd': 12, 'z': 10}
>>>
```

说明:

(1) choice()函数返回一个列表、元组或者字符串中的一个随机项,本例中会随机产生 26 个小写字母中的 1 个赋给变量 r。

(2) dict1[r]=dict1.get(r,0)+1 是本例的核心语句,r 是一个随机的字母,对以 r 为键的字典 dict1 进行赋值(如果没有 r 键,则添加字典元组),dict1.get(r,0)会返回以 r 为键的值,如果 r 键不存在,则返回 0,假设 r 随机产生的字母序列为'a'、'b'、'a'、'c'、'a',则执行顺序如表 4.3 所示。

表 4.3　语句执行实例

r产生顺序	语　　句	结　　果
a	dict1['a']=dict1.get('a',0)+1	字典中无'a'键,添加'a':1 元素
b	dict1['b']=dict1.get('b',0)+1	字典中无'a'键,添加'b':1 元素
a	dict1['a']=dict1.get('a',0)+1	字典中已有'a'键,取出键为'a'的元素的值 1 再加 1,元素更新为'a':2
c	dict1['c']=dict1.get('c',0)+1	字典中无'c'键,添加'c':1 元素
a	dict1['a']=dict1.get('a',0)+1	字典中已有'a'键,取出键为'a'的元素的值 2 再加 1,元素更新为'a':3
…	…	…



4.5　集合

集合是无序的可变序列,集合元素放在一对大括号间(和字典一样),元素之间用逗号分隔。在一个集合中,元素不允许重复。集合的元素类型只能是固定的数据类型,如整型、字符串、元组等,而列表、字典等是可变数据类型,不能作为集合中的数据元素。例如下面的集合都是合法的:

```
{1,2,3,4,5,6,7}
{"a","b","c","d","e"}
{(1, 2), (1, 3), (2, 1), (2, 3), (2, 2), (1, 1)}
```

4.5.1　创建集合

创建可变集合可以使用赋值语句或者 set()函数创建;创建不可变集合可以使用 frozenset()创建。

1. 通过赋值语句创建集合

```
>>>set1={1,2,3,4,3,2,1}                    #创建集合对象时,重复元素自动去掉
>>>set1
{1, 2, 3, 4}
>>>set2={(1,1),(1,2),(2,1),(2,2)}
```

```
>>>set2
{(1, 2), (1, 1), (2, 1), (2, 2)}
```

2. 通过 set()函数创建可变集合

```
>>>set3=set()  #赋值语句 set3={}创建的是空字典,空集只能通过没有参数的 set()函数创建
>>>set3
set()
>>>set4=set([1,2,3,4,3,2,1])            #将列表对象创建为集合
>>>set4
{1, 2, 3, 4}
>>>set5=set((1,2,3,4,3,2,1))            #将元组对象创建为集合
>>>set5
{1, 2, 3, 4}
```

3. 通过 frozenset()创建不可变集合

```
>>>set6=frozenset([1,2,3,4,3,2,1])          #不可变集合创建后不可更新
>>>set6
frozenset({1, 2, 3, 4})
>>>
```

4.5.2 访问集合

集合中的元素是无序的,而且也没有任何的键与集合元素对应,所以无法取指定的某个元素,只能遍历整个集合访问其中的元素。例如:

```
>>>set1={1,7,2,6,5,4,3,7,1}
>>>for i in set1:
            print(i,end=',')


    1,2,3,4,5,6,7,
>>>
```

4.5.3 更新集合

集合的更新只有添加元素和删除元素两种操作，而且只有可变集合是可以更新的，不可变集合创建后不能更新。

1. 添加元素

通过 add()函数可以向集合中添加一个元素；通过 update()函数可以向集合中添加多个元素。在向集合中添加元素时，如果新元素与集合中原有的元素重复，将不会被添加。例如：

```
>>>set1={1,2,3,4}
>>>set1.add(5)                #add()函数一次只能向集合中添加一个元素
>>>set1
{1, 2, 3, 4, 5}
>>>set1.add(4)                #如果添加的元素重复,则新元素不会被添加
>>>set1
{1, 2, 3, 4, 5}
>>>set1.update({1,2,6,7})     #update()函数的实质是将一个新集合合并到原有集合中
>>>set1
{1, 2, 3, 4, 5, 6, 7}
>>>
```

2. 删除元素

删除一个集合中的元素可以使用 remove()函数、pop()函数、discard()函数或者 clear()函数。例如：

```
>>>set1={1,2,3,4,5,6,7}
>>>set1.remove(4)                #删除集合元素 4
>>>set1
{1, 2, 3, 5, 6, 7}
>>>set1.remove(8)                #当删除的元素不在集合中时,返回错误
Traceback (most recent call last):
File "<pyshell#145>", line 1, in<module>
    set1.remove(8)
KeyError: 8
>>>set1.discard(2)               #discard函数和 remove函数的使用方法类似
```

```
>>>set1
{1, 3, 5, 6, 7}
>>>set1.discard(4)                  #当删除的元素不在集合中时,discard函数不报错
>>>set1
{1, 3, 5, 6, 7}
>>>set1.pop()                       #随机删除一个集合元素
3
>>>set1
{1, 5, 6, 7}
>>>set1.clear()                     #清除集合中的所有元素
>>>set1
set()
>>>
```

4.5.4 集合常用的其他操作

Python 语言中的集合和数学中的集合概念类似,也可以进行并、交等运算。表 4.4 列出了集合的常用操作符和函数。

表 4.4 常用集合操作

操　作	功　能
x in set1	元素 x 在 set1 中,返回 True,否则返回 False
x not in set1	元素 x 不在 set1 中,返回 True,否则返回 False
set1==set2	set1 与 set2 元素个数,元素值完全相同,返回 True,否则返回 False
set1!=set2	set1 与 set2 不同,返回 True,否则返回 False
set1<set2	set1 是 set2 的真子集,返回 True,否则返回 False
set1<=set2	set1 是 set2 的子集,返回 True,否则返回 False
set1 & set2	求 set1 与 set2 的交集
set1 | set2	求 set1 与 set2 的并集
set1-set2	求 set1 减 set2,即求属于 set1 不属于 set2 的元素集合
set1^set2	求 set1 和 set2 的对称差
len(set1)	求 set1 的元素个数



习题 4

一、填空题

1. 假设列表对象 list1 的值为[3, 4, 5, 6, 7, 9, 11, 13, 15, 17],那么切片 list1[3:7]得到的值是________。

2. 使用列表推导式生成包含 100 以内奇数的列表,语句可以写为________。

3. 任意长度的 Python 列表、元组和字符串中最后一个元素的下标为________。

4. 字典中多个元素之间使用________分隔开,每个元素的键与值之间使用________分隔开。

5. 字典对象的________方法返回字典的键列表,________方法返回字典的值列表,________方法返回字典的元素列表。

6. 已知字典 x={i:(i+3) * * 2 for i in range(5)},那么表达式 sum(x.values()) 的值为________。

7. 表达式 set([3,2,3,1])=={1, 2, 3} 的值为________。

二、判断题

1. 列表、元组、字符串是 Python 的有序序列。 ( )

2. 元组、列表、字典都是有序的数据结构。 ( )

3. Python 集合中的元素不允许重复。 ( )

4. 在 Python 中,运算符+不仅可以实现数值的相加和字符串的连接,还可以实现集合的并集运算。 ( )

5. 已知 A 和 B 是两个集合,并且表达式 A<B 的值为 False,那么表达式 A>B 的值一定为 True。 ( )

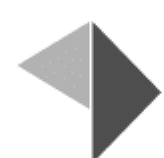
6. 无法删除集合中指定位置的元素,只能删除特定值的元素。 ( )

第 5 章

字符串和正则表达式

学习目标

- 了解 Python 的字符串运算
- 熟悉字符串的格式化、索引和分片的具体方法
- 掌握 Python 中字符串的基本运算符
- 掌握 Python 中字符串的运算函数
- 掌握 Python 中字符的串运算方法
- 掌握正则表达式的使用



5.1 字符串的基本操作

Python 中的字符串是一种非常重要的数据类型，它支持丰富的操作和运算。Python 的字符串可以看作是一串连续存储的字符的序列，它可以通过索引进行顺序访问。

5.1.1 字符串的格式化

字符串的格式化通常用在 print()函数中，用来实现输出字符的特定样式。格式说明符和普通字符构成一个格式字符串，通过格式运算符%限定输出数据的显示样式。

格式字符串的格式如下：

```
<格式字符串>%(<值 1>,<值 2>,…,<值 n>)
```

通过格式运算符实现将一个值插入到格式字符串中相应运算符%出现的位置。

```
>>>print("今天是%d年%d月%d日,天气%s!"%(2017,5,16,'晴'))
今天是 2017年 5月 16日,天气晴!
>>>
```

表 5.1　常用的格式说明符

符　号	描　　述
%c	字符及其 ASCII 码
%s	字符串
%d	十进制整数
%o	八进制整数
%x	十六进制整数(用小写字母)
%X	十六进制整数(用大写字母)
%f	浮点数字,可指定小数点后的精度
%e	浮点数字,科学计数法,用小写 e
%E	浮点数字,科学计数法,用大写 E
%g 或%G	浮点数字,根据值采用不同模式

5.1.2　字符串的索引与分片

1. 索引

字符串中的字符按位置进行了编号,称为索引,使用时可以通过这个编号访问字符串中的特定字符。字符串的第一个字符的编号为 0,一个长度为 L 的字符串的最后一个字符编号为 L-1。例如,可以通过以下方式访问指定字符:

```
>>>str="God Wants To Check The Air Quality"
>>>str[0],str[1],str[19]
('G', 'o', 'T')
```

Python 同时允许根据索引反向访问字符串,此时字符串的编号从-1 开始。例如:

```
>>>str="God Wants To Check The Air Quality"
>>>str[-1],str[-13],str[-26]
('y', 'e', 's')
>>>
```

2. 分片

字符串的分片指通过索引对字符串进行切片的操作。分片操作格式如下：

```
<字符串名>[i:j:k]
```

这里的 i 表示起始编号，j 表示结束编号，k 表示编号增加步长。注意，切片的位置不包含 j 位置上的字符。例如：

```
>>>str="God Wants To Check The Air Quality"
>>>str[0:8:2]              #将 str 字符串第 0、2、4、6 的位置进行切片
'GdWn'
```

分片语句中 i、j、k 均可以省略。i 省略时，表示从 0 或 -1 开始；j 省略时，表示到最后一个字符；k 省略时，表示步长为 1。例如：

```
>>>str="God Wants To Check The Air Quality"
>>>str[4:18]
'Wants To Check'
>>>str[::2]
'GdWnsThc h iult'
>>>str[27::]
'Quality
```

【例 5.1】 利用字符串分片操作，逆序输出字符串。

程序代码如下：

```
#example5.1
st=input('输入一个字符串：')
print("原字符串：%s"%(st))
print("逆序字符：%s"%(st[-1::-1]))
```

程序的运行结果如下：

```
>>>
================RESTART:C:/Users/Python/5-1.py================
输入一个字符串：12345678
原字符串：12345678
```

```
逆序字符：87654321
>>>
```

说明：

(1) 程序用来实现将输入的字符串逆序输出。

(2) 通过分片生成逆序字符串，st[-1::-1]表示从末尾进行分片直到完成，步长为-1。

【例 5.2】 查询月份英文缩写。

```
#example5.2
#查询英文月份
st='''一月 Jan 二月 Feb 三月 Mar 四月 Apr 五月 May 六月 Jun
七月 Jul 八月 Aug 九月 Sep 十月 Oct 十一月 Nov 十二月 Dec'''
mon=int(input('input a month:'))
n=(mon-1)*5
print('%s的英文简称为：%s'%(st[n:n+2],st[n+2:n+5]))
```

程序的运行结果如下：

```
>>>
================RESTART:C:/Users/Python/5-2.py================
input a month:6
六月的英文简称为：Jun
>>>
```

说明：

(1) 程序用来实现在字符串 st 中分片生成月份的英文缩写。

(2) st 中用固定格式顺序存放 12 个月的英文缩写形式。

(3) 为方便定位，将每个月份的信息长度固定为 5(3 个英文和 2 个汉字)。

(4) 通过表达式 n=(mon-1)*5 的计算，得到相应月份所在的起始位置。

(5) 分片 st[n:n+2]生成月份字符串，st[n+2:n+5])生成英文缩写字符串。

5.1.3　字符串的基本运算

Python 支持通过一些运算符实现几个字符串的基本运算，包括字符串的连接、判断子串、字符串的比较等。字符串的基本运算符如表 5.2 所示。

表 5.2 字符串基本运算符及功能

运算符	功能
s1+s2	连接字符串 s1 和 s2
s1 * n	生成由 n 个 s1 组成的字符串
s1 in s2	如果 s1 是 s2 的子串,返回 True,否则返回 False
>,<,==	比较字符串的 ASCII 码,多个字符时从左向右依次比较

字符串的运算实例：

```
>>>s1='Python 程序设计'
>>>s2='入门'
>>>n=2
>>>s1+s2
'Python 程序设计入门'
>>>s1*n
'Python 程序设计 Python 程序设计'
>>>s2 in s1
False
>>>'a'>'A'                          #单个字符的比较 ASCII 码的值,a 为 97,A 为 65
True
>>>'this is a test'>'this is '      #多个字符的比较,从左向右依次每个字符
True
>>>
```

5.1.4 字符串运算函数

除了字符串运算符以外,Python 还支持一些字符串运算函数。常用的几个字符串运算函数如表 5.3 所示。

表 5.3 字符串运算函数及功能

函数名	功能	函数名	功能
len(s)	返回集合长度	ord(s)	返回一个字符的 ASCII 码
chr(x)	返回整数 x 对应的 ASCII 字符	str(x)	将数字转换为字符串

如下例所示：

```
>>>len('Python 程序设计')
11
>>>print(ord('a'),ord('A'))
97 65
>>>print(chr(66),chr(98))
B b
>>>'10'+str(3.1415926)
'103.1415926'
>>>
```

【例 5.3】 电文加密程序。

```
#example5.3
#电文加密
original=input('输入原文: ')
cryption=''
for s1 in original:
    if s1.isalpha():
        i=ord(s1)+5
        if s1.isupper():
            if i>ord('Z'):i-=26
        else:
            if i>ord('z'):i-=26
        s2=chr(i)
        cryption+=s2
    else:
        cryption+=s1
print('输出密文: %s'%(cryption))
```

程序的运行结果如下：

```
>>>
================RESTART:C:/Users/Python/5-3.py=================
输入原文:Windows!
输出密文:Bnsitbx!
>>>
```

说明：

(1) 电文加密规则：将原文中的字母转换为英文字母表中其后面第 5 个字母，例如，

A->F。要求保持原文的大小写状态,且除字母外的其他字符不变,不进行加密处理。

(2) 代码中 s1.isalpha()字符串方法,判断 s1 是字母返回 True,否则返回 False。

(3) 代码中 s1.isupper()字符串方法,判断 s1 是大写字母返回 True,否则返回 False。

(4) 字母在 ASCII 表中分别按"a"～"z"和"A"～"Z"的顺序排列,即 26 个字母的 ASCII 码是连续的。"a"～"z"的 ASCii 码是 97～122,"A"～"Z"的 ASCII 码是 65～90。

5.1.5 字符串运算方法

Python 是面向对象的程序设计语言,Python 中的所有数据类型都是一个类。类的方法就是封闭在类中的函数,与内置的功能相似,但是调用方法不同。字符串对象通过一些常用方法完成相应的运算。调用字符串方法的格式如下:

```
<字符串名>.<方法名>(<参数>)
```

字符串常用方法如表 5.4 所示。

表 5.4 字符串常用方法及功能

方法名	功能
s.lower()	将字符串转换为小写字母
s.upper()	将字符串转换为大写字母
s.capitalize()	将字符串转换为首字母大写
s.tiltle()	将第一个字符转换为大写
s.replace(old,new[,count])	将 s 中的 old 字符串替换为 new,count 为替换的次数
s.split([sep,[maxsplit]])	以 sep 为分隔符将 s 拆分为一个列表,默认分隔符为空格,maxsplit 表示拆分的次数,默认为-1,表示无限制
s.find(s1[,start,[end]])	返回 s1 在 s 中出现的位置。如果没有出现则返回-1
s.count(s1[,start,[end]]	返回 s1 在 s 中出现的次数
s.isalnum()	判断 s 是否为全字母和数字,且至少一个字符
s.isalpha()	判断 s 是否为全字母,且至少一个字符
s.isupper()	判断 s 是否为全大写字母
s.islower()	判断 s 是否为全小写字母
s.format()	格式输出字符串(用法与格式运算符%类似)

字符串大小写转换方法的实例：

```
>>>s='this is a test!'
>>>s.upper()
'THIS IS A TEST!'
>>>s.capitalize ()
'This is a test!'
>>>s.title()
'This Is A Test!'
>>>
```

字符串拆分方法的实例：

```
>>>s='this is a test!'
>>>s.split()                          #将字符串以空格为分隔符拆分
['this', 'is', 'a', 'test!']
>>>s.split(sep=' ',maxsplit=1)        #将字符串以空格为分隔符拆分一次
['this', 'is a test!']
>>>
```

字符串替换和查找方法的实例：

```
>>>s='this is a test!'
>>>s.find('t')                        #查找字母 t 在字符串 s 中第一次出现的位置
0
>>>s.count('t')                       #查找字母 t 在字符串 s 中出现的次数
3
>>>s.replace('t','T',2)               #将字符串中的 t 替换 T,替换 2 次
'This is a Test!'
>>>
```

字符串格式化方法的实例：

```
a='this is %s %s' % ('an', 'apple')                          #显示结果为 this is an apple
a='this is {} {}'. format ('apple', 'an')                    #显示结果为 this is apple an
a='this is {1} {0}'. format ('apple', 'an')                  #显示结果为 this is an apple
a='this is {number} {fruit}'. format (number='an', fruit='apple')
                                                             # 显示结果为 this is an apple
```

【例 5.4】 统计文章中单词出现的次数。

程序代码如下：

```
#example5.4
passage='Do not trouble trouble till trouble troubles you.'
word=input('input a word:')
n=passage.count(word)
print('%s出现的次数:%s'%(word,n))
```

程序的运行结果如下：

```
>>>
================RESTART:C:\Users\Python • -4.py================
input a word:trouble
trouble出现的次数:4
>>>
```



5.2 正则表达式的使用

正则表达式(Regular Expression)描述了一种字符串匹配的模式,用来检查一个字符串是否含有某种子串、将匹配的子串进行替换或者从某个串中取出符合某个条件的子串等。正则表达式是由普通字符(例如字符 a 到 z)以及特殊字符(称为元字符)组成的文字模式。正则表达式作为一个模板,将某个字符模式与所搜索的字符串进行匹配。

Python 中的常用元字符及其含义如表 5.5 所示。

表 5.5 常用元字符及其含义

元字符	描述
.	匹配除换行符以外的任意一个字符
?	重复匹配 ? 前面字符 0 次或 1 次
*	重复匹配 * 前面字符 0 次或多次
+	重复匹配+前面字符 1 次或多次
|	匹配|前面或后面字符
^	匹配以^后面字符开头的字符串
$	匹配以 $ 前面字符结尾的字符串

续表

元 字 符	描　　述
\	表示一个转义字符
[]	匹配[]内的任意一个字符，如[ab]c 匹配"ac"、"bc"
[^]	匹配不在[]中的任意一个字符，如[^ab]c 匹配除 a、b 之外的字符与 c 的组合
{n,m}	重复匹配 n-m 次，如{n,}重复 n 次或多次，{n}表示重复 n 次
\w	匹配字母数字下画线，等价于[a-z A-Z 0-9_]
\W	匹配非字母数字下画线
\s	匹配任意空白字符，等价于[\t\n\r\f]
\S	匹配任意非空字符
\d	匹配任意数字，等价于[0-9]
\D	匹配任意非数字
\b	匹配一个单词边界，指单词的词首和词尾。如'er\b' 可以匹配"never" 中的 'er'，但不能匹配 "verb" 中的 'er'
\B	匹配非单词边界。如'er\B'匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'

1. 正则表达式实例

国内电话号码："024-86[0-9]{6}"，沈阳地区以 024-86 开头的电话号码。如 024-86456789。

国内邮政编号："[1]{2}[0-9]{4}"，辽宁省内的邮政编码(以"11"开头)。如 110030。

身份证号码："\d{15}|\d{18}"，任意 15 位或 18 位的号码。

网站格式："(http://)?(www\.)?synu\.edu\.cn"，沈阳师范大学的官网。可匹配的网址有 http://www.synu.edu.cn，www.synu.edu.cn，http://synu.edu.cn，synu.edu.cn。

特定格式字符串：'^[a-zA-Z]{1}([a-zA-Z0-9_]){4,19}$'，匹配必须以字母开头，由字母、数字、下画线组成的长度为 5～20 的字符串。

邮件地址：'^\w+@(\w+\.)+\w+$'，可匹配合法电子邮件地址。

IP 地址：'^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$'，可匹配合法的 IP 地址。

2. 使用正则表达式的方法

(1) 直接使用 re 模块

Python 中正则表达式可以通过 re 模块使用，re 模块可以检查所给的字符串是否与

指定的正则表达式相匹配,它支持多种字符串的运算函数。

【例 5.5】 re 模块使用正则表达式。

程序代码如下:

```
#example 5.5
import re
s='Do not trouble trouble till trouble troubles you.'
r='t[a-zA-Z]+'
print(re.match(r,s))                #在 s 开始位置匹配正则表达式
print(re.search(r,s))               #在 s 任意位置匹配正则表达式
print(re.findall(r,s))              #以列表形式返回 s 中全部匹配字符串
```

程序运行后,匹配的结果如下:

```
>>>
=============RESTART: C:/Users/ Python /example5-5.py=============
None
<_sre.SRE_Match object; span=(7, 14), match='trouble'>
['trouble', 'trouble', 'till', 'trouble', 'troubles']
>>>
```

(2) 使用正则表达式对象

首先使用 re 模块的 compile()方法将正则表达式编译生成正则表达式对象,然后再使用正则表达式对象的方法进行字符串处理。使用编译后的正则表达式对象不仅可以提高字符串的处理速度,还具有更加强大的字符串处理功能。

【例 5.6】 使用正则表达式对象。

程序代码如下:

```
#example 5.6
import re
s='Do not trouble trouble till trouble troubles you.'
r=re.compile('t[a-zA-Z]+')
print(r.findall(s))         #以列表形式返回 s 中的全部匹配字符串
print(r.sub('-',s))         #替换匹配字符串
```

程序运行后,匹配的结果如下:

```
>>>
============RESTART: C:/Users/ Python /example5-6.py============
['trouble', 'trouble', 'till', 'trouble', 'troubles']
Do not tell telltelltelltell you.
>>>
```

【例 5.7】 使用正则表达式对象的方法 split()。

程序代码如下：

```
#example 5.7
import re
s='210.30.208.7'
r=re.compile('。')
print(r.split(s))                    #按正则表达式分割字符串
print(r.split(s,1))                  #按正则表达式分割字符串,分割次数为 1
```

程序运行后,匹配的结果如下：

```
>>>
============RESTART: C:/Users/ Python /example5-7.py============
['210', '30', '208', '7']
['210', '30.208.7']
>>>
```

习题 5

一、填空题

1. 表达式 len('Python 程序设计')的值为________。
2. 已知 x='I love Python! ',则表达式 x[7:]+x[:7]的值为________。
3. 已知 x='I love Python! ',表达式 'love' in x 的值为________。
4. 已知 x='I love Python! ',表达式 'python' in x 的值为________。
5. 表达式 chr(ord('A')+5) 的值为________。
6. 表达式 'Hello Python!'. find('Python')的值为________。
7. 表达式 'Hello Python!'. count('h') 的值为________。

8. 表达式 'Hello Python!'.find('p') 的值为________。

9. 表达式 'Hello Python!'.find('Py')的值为________。

10. 表达式'www.cctv.com'.split('.') 的值为________。

11. 表达式 'Hello world!'.upper() 的值为________。

12. 已知 x='a234b123c',并且 re 模块已导入,则表达式 re.split('\d+', x) 的值为________。

13. 已知 st='Hello Python!',表达式 st[-5]的值为________。

14. 已知 st='Hello Python!',表达式 st[-4::1]的值为________。

15. 已知 st='Hello Python,Hello World!',表达式 st.replace('Hello','Love')的值为________。

16. 已知 st='Hello Python,Hello World!',表达式 st.replace('Hello','Love',1)的值为________。

17. 正则表达式元字符________表示重复匹配前面字符一次或多次。

18. 正则表达式元字符________表示匹配任意数字。

19. 正则表达式元字符________用来表示匹配字母、数字、下画线。

20. 模块 re 的________方法用来编译正则表达式对象。

21. 模块 re 的________方法用来在字符串开始处进行匹配。

22. 模块 re 的________方法可以以列表形式返回全部匹配字符串。

23. 表达式 re.split(':\/+', 'http://www.cctv.com')的值为________。

24. 假设 re 模块已导入,那么表达式 re.findall('\d{1,3}','210.30.208.7')的值为________。

25. 表达式 'Hello world!'.count('o')的值为________。

二、判断题

1. 正则表达式 search()方法是在整个字符串中寻找模式,匹配成功则返回对象,匹配失败则返回空值 None。 ( )

2. 字符串方法 s.isalnum(),判断 s 是否为全数字,且至少一个字符。 ( )

3. 正则表达式元字符^用来表示从字符串开始处进行匹配,而[^]表示反向匹配,不匹配方括号中的字符。 ( )

4. 正则表达式元字符\s 用来匹配任意空白字符。 ( )

5. 正则表达式元字符\D 用来匹配任意数字字符。 ( )

第 6 章

Python 函数和模块

学习目标

- 掌握函数的定义和调用方法
- 理解函数参数的传递方式
- 理解位置参数、关键字参数、默认值参数和可变参数
- 了解函数返回值的含义
- 了解函数嵌套的原理
- 理解函数递归调用的方法
- 掌握安装第三方模块的方法
- 了解 turtle 和 jieba 两个第三方模块的基本使用方法



6.1 函数的定义

在软件开发的过程中，经常有很多操作是完全相同或者相似的，如果重复使用相同或者相似的代码实现功能，将造成代码阅读、理解和维护的难度增加，同时也不利于程序的调试和纠错，因此应当尽量减少这种用法。在各种程序设计语言中，通常可以采用函数解决这一问题。函数是将可以被反复使用的、用来实现单一或相关联功能的代码段封装或组织在一起，形成一个独立的程序单位，并为该程序单位定义一个相应的名称，这样的程序单位称为函数，该名称称为函数名。利用函数可以提高应用程序的模块性和代码的重复利用率。在第 2.4 节介绍的 Python 常用内置函数就是由系统事先定义好的、可以被用户直接且重复使用的程序单位，称为系统函数。但这还远远不够，为了实现对各种复杂情况下的代码复用，Python 还可以由用户建立自定义函数。为了更好地管理大规模且反复被重用的大段程序代码，Python 引入了模块的概念，将大段程序或多个函数段封装为一个文件，可以反复使用，进一步地提出了包的概念。

在 Python 中,定义函数的语法如下:

```
def<函数名>([参数列表]):
    <函数体>
```

其中,def 关键字用来定义函数,函数名是由用户定义的任何有效的标识符,函数名后面的圆括号必须有,其中参数列表中的参数由变量充当,可以有多个参数,参数之间用逗号分隔。在函数体内该参数没有确定的值,只有在调用程序和调用函数时才通过形参向函数传递值,因此该参数被称为形式参数,简称形参。当程序调用该函数时,通过写在调用函数括号内部的具体参数向形参传递相应的值,这时括号内部的参数被称为实际参数,简称实参。实参可以是常量、变量或者表达式。在调用函数时,实参的值必须是确定的,该值将以参数传递的形式传递给函数中的形参。函数定义时,括号后面的冒号是必不可少的,同时,函数体是至少有一条语句的代码段,必须保持与 def 关键字有一定的空格缩进,函数体中的代码段在函数被调用时执行。

【例 6.1】 函数的定义。

程序代码如下:

```
#example6.1
def MyFirstFunction():#定义函数 MyFirstFunction()
    print("这是第一个函数")
    print("我非常喜欢学习 Python")
MyFirstFunction()#调用函数 MyFirstFunction()
```

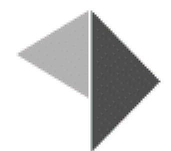
程序运行结果如下:

```
>>>
=====================RESTART:C:\Python\example6.1.py==============
这是第一个函数
我非常喜欢学习 Python
>>>
```

说明:

(1) 用 def 定义函数 MyFirstFunction()要注意区分字母大小写,函数名后面的括号不可省略;

(2) 在主程序中直接通过函数名调用函数,函数名后面也必须有括号,调用函数的过程就是执行函数中语句的过程。



6.2　函数的调用

在 Python 中，函数调用要在函数定义之后进行，具体格式如下：

```
<函数名>(<参数列表>)
```

其中，函数名为前面 def 语句定义的函数名，参数列表中的参数必须有确定的值，称为实际参数，简称实参，实参也可以由多个参数组成，中间用逗号分隔。

在具体的程序语句中，调用函数可以采用如下三种方式。

(1) 直接调用函数名，此时将函数语句完整执行一遍，调用程序往往不需要函数的返回值。

(2) 函数作为表达式的一部分出现，此时调用程序将使用函数的返回值，并参与表达式的数据计算。

(3) 函数被嵌套在另一个函数中，函数的返回值被当作另外函数的实参使用。

【例 6.2】 设计一个求阶乘的函数，在主程序中输入一个值，调用该函数，求得该值的阶乘并输出。

程序代码如下：

```
#example6.2
def factroial(x):
    n=1
    for i in range(1,x+1):
        n=n*i
    return n
y=int(input("请输入一个正整数:"))
if y<0:
    print("抱歉,输入错误!")
else:
    print(y,"的阶乘是:",factroial(y))
```

三次运行程序，分别输入不同的数据，结果如下：

```
>>>
=================RESTART:C:\Python\example6.2.py=================
请输入一个正整数:6
```

```
6的阶乘是:720
>>>
=================RESTART:C:\Python\example6.2.py==================
请输入一个正整数:5
5的阶乘是:120
>>>
==================RESTART:C:\Python\example6.2.py=================
请输入一个正整数:-6
抱歉,输入错误!
>>>
```

说明:

(1) 定义自定义函数 factroial(),其中 x 为形参,用来接收函数调用时传递过来的实参。

(2) 定义函数后,在主程序中调用该函数,将实参 y 的值传递给形参 x。

【例 6.3】 编写一个根据圆的半径求圆的面积的函数,并利用该函数计算圆的面积。

程序代码如下:

```
#example6.3
def area(x):
    s=3.14159*x*x
    return s
r=eval(input("请输入圆的半径:"))
print("圆的面积为:",area(r))
```

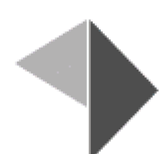
运行程序,并输入圆半径,检验函数的功能,结果如下:

```
>>>
==================RESTART:C:\Python\example6.3.py=================
请输入圆的半径:2.5
圆的面积为:19.6349375
>>>
```

说明:

(1) 定义自定义函数 area(),其中 x 为形参,用来接收函数调用时传递过来的实参。

(2) 定义函数后,在主程序中调用该函数,将实参 r 的值传递给形参 x。



6.3　函数的参数和返回值

6.3.1　参数传递的方式

当 Python 中的主程序调用函数时，实参的值传递给形参，实际上是将实参所指向的对象的地址传递给了形参。因此，如果传递的对象是不可变对象(如数值、字符、元组等)，在函数体中形参值的变化不会影响到实参。如果传递的对象是可变对象(如列表、字典等)，在函数中形参值的变化会影响到实参。

【例 6.4】 传递不可变对象，形参的变化不会影响到实参。

程序代码如下：

```
#example6.4
def add(x):
    print("形参 x 的初始值是: ",x)
    x+=1
    print("形参 x 的最终值是: ",x)
y=4
add(y)
print("实参 y 的值是: ",y)
```

程序运行结果如下：

```
>>>
================== RESTART:C:\Python\example6.4.py=================
形参 x 的初始值是: 4
形参 x 的最终值是: 5
实参 y 的值是: 4
>>>
```

说明：

(1) 主程序中定义了一个变量 y，初值为 4，即 y 指向数值 4。

(2) 主程序中调用 add()函数，变量 y 作为实参传递给形参 x，此时 x 也指向数值 4。

(3) 在函数体内，给 x 赋值为数值 5，因为数值是不可变对象，所以 x 指向了新的对象——数值 5；此时，作为实参的变量 y 仍然指向原数值 4，即 y 的值不变。

【例 6.5】 传递可变对象，形参的变化会影响到实参。

程序代码如下：

```
#example6.5
def change(v):
    v.append(3)
    print(v)
m=[1]
change(m)
print(m)
```

程序运行结果如下：

```
>>>
=================RESTART:C:\Python\example6.5.py==================
[1,3]
[1,3]
>>>
```

说明：

(1) 主程序中定义了一个变量 m,m 指向列表[1]。

(2) 主程序中调用 change()函数,变量 m 作为实参传递给形参 v,此时 m 和 v 都指向列表[1]。

(3) 在函数体内,给形参 v 执行 append 操作,因为列表是可变对象,所以该操作直接作用在原来的列表上,原列表变为[1,3],此时并不会生成新的对象,m 和 v 都指向列表[1,3]。

【例 6.6】 字典作为形参,形参的变化会影响到实参。

程序代码如下：

```
#example6.6
def change(d):
    d['姓名']='诸葛亮'
    d['年龄']=59
    d['性别']='男'
a={'姓名':'林黛玉','性别':'女','年龄':20}
print(a)
change(a)
print(a)
```

程序运行结果如下：

```
>>>
=================RESTART:C:\Python\example6.6.py=================
{'姓名': '林黛玉', '性别': '女', '年龄': 20}
{'姓名': '诸葛亮', '性别': '男', '年龄': 59}
>>>
```

说明：

(1) 调用函数 change()时，字典变量 a 作为实参传递给形参 d，a 与 d 指向同一个列表。

(2) 调用函数 change()后，字典变量 a 的值发生了变化，因为字典变量是可变对象，因此形参的变化直接影响了实参。

6.3.2　位置参数和关键字参数

1. 位置参数

默认情况下，Python 要求调用函数时参数的个数、位置和顺序要与函数定义中的形参一致，这种参数也被称为位置参数。

【例 6.7】 位置参数的使用。

程序代码如下：

```
#example6.7
def posi_args(a,b):
    print("第一个参数的值是: ",a)
    print("第二个参数的值是: ",b)
posi_args(100,200)
posi_args(200,100)
```

程序运行结果如下：

```
>>>
=================RESTART:C:/Python/example6.7.py=================
第一个参数的值是: 100
第二个参数的值是: 200
第一个参数的值是: 200
第二个参数的值是: 100
>>>
```

说明：调用函数 posi_args()时，实参的值按照形参的位置一一对应地传递给形参，所以调用时输入的实参顺序不同，得到的结果也不同。

【例 6.8】 修改例 6.7 中的函数调用语句，查看程序运行结果。

(1) 调用 posi_agrs()函数时，实参数量少于形参数量

程序代码如下：

```
#example6.8-a
def posi_args(a,b):
    print("第一个参数的值是：",a)
    print("第二个参数的值是：",b)
posi_args(100)
```

(2) 调用 posi_agrs()函数时，实参数量多于形参数量

程序代码如下：

```
#example6.8-b
def posi_args(a,b):
    print("第一个参数的值是：",a)
    print("第二个参数的值是：",b)
posi_args(200,300,400)
```

运行以上两个程序，都会产生错误，第一个程序产生的错误如下：

```
>>>
================RESTART:C:/Python/example6.8-a.py================
Traceback (most recent call last):
  File "C:/Python/example6.7-2.py", line 5, in<module>
    posi_args(100)
TypeError: posi_args() missing 1 required positional argument: 'b'
>>>
```

第二个程序产生的错误如下：

```
>>>
================RESTART:C:/Python/example6.8-b.py================
Traceback (most recent call last):
```

```
  File "C:/Python/example6.7-2.py", line 5, in<module>
    posi_args(200,300,400)
TypeError: posi_args() takes 2 positional arguments but 3 were given
>>>
```

说明：函数中定义有两个位置参数 a 和 b，调用时实参的数量与形参的数量必须相同，无论实参的数量少于形参还是实参的数量多于形参，Python 都会报错。

2. 关键字参数

在调用函数时，可以明确指定参数值传递给哪个形参，这样的参数被称为关键字参数。使用了关键字参数，可以不考虑形参与实参的位置和顺序的一一对应。

【例 6.9】 关键字参数，参数位置不必一一对应。

程序代码如下：

```
#example6.9
def keywords(a,b):
    print("传递给参数 a 的值为",a)
    print("传递给参数 b 的值为",b)
keywords(100,200)
keywords(b=200,a=100)
keywords(a=100,b=200)
```

程序运行结果如下：

```
>>>
=================RESTART:C:\Python\example6.9.py=================
传递给参数 a 的值为 100
传递给参数 b 的值为 200
传递给参数 a 的值为 100
传递给参数 b 的值为 200
传递给参数 a 的值为 100
传递给参数 b 的值为 200
>>>
```

说明：

(1) 主程序定义函数 keywords()，然后三次调用该函数，三次调用的结果都相同。

(2) 第一次调用为普通调用方式，按照顺序和位置传递参数，100 传递给形参 a，

200 传递给形参 b。

(3) 后两种调用方式 keywords(b=200,a=100)和 keywords(a=100,b=200)使用了关键字参数,虽然参数书写的顺序不同,但两条语句都明确地把 100 传递给了形参 a,把 200 传递给了形参 b,故得到的结果是相同的,也就是说,使用关键字参数调用函数时,可以打乱参数传递的顺序。

6.3.3 默认值参数

如前所述,一般情况下,实参的个数与形参的个数相等且位置是一一对应的,但在某些特殊情况下,实参也可以少于形参,如设置了默认值参数。Python 允许在创建函数时为形参指定默认值。调用函数时,可以不为设置了默认值的形参传递值,此时将使用函数定义时形参的默认值,也可以通过显式赋值方式改变默认值。指定了默认值的形参也被称为可选参数,没有指定默认值的形参被称为必选参数。带有默认值参数的函数按如下格式定义:

```
def<函数名>(…<形参=默认值>…)
    <函数体>
```

【例 6.10】 定义求 X^n 的函数,通过函数默认值,设定该函数在默认情况下求 X^2。

程序代码如下:

```
#example6.10
def power(x,n=2):                        #定义默认值参数 n
    s=1
    for i in range(1,n+1):
        s=s * x
    return s
print(power(5))                          #形参 n 的默认值为 2,求 5 的平方
print(power(6))                          #形参 n 的默认值为 2,求 6 的平方
print(power(5,3))                        #改变默认值参数 n 的值,求 5 的 3 次方
print(power(3,4))                        #改变默认值参数 n 的值,求 3 的 4 次方
```

程序运行结果如下:

```
>>>
================RESTART:C:\Python\example6.10.py================
```

```
25
36
125
81
>>>
```

说明：

(1) 本例的函数 power()中,定义了两个参数 x 和 n。

(2) 形参 n 的默认值为 2,调用函数时,可以不为 n 传递值,将求任意数的平方,如 power(5)。

(3) 调用函数时也可以为形参 n 传递不同的值,从而改变默认值参数,此时函数的功能为求任意数的 n 次方,如 power(5,3)为求 5 的 3 次方。

【例 6.11】 多个默认值参数的使用。

程序代码如下：

```
#example6.11
def add_three(x,y=4,z=5):        #指定两个默认值参数 y 和 z
    s=x+y+z
    return s
print(add_three(3))              #只给必选参数 x 传递值,可选参数 y 和 z 使用默认值
print(add_three(7,8,9))          #给必选参数 x、可选参数 y 和 z 都传递值
```

程序运行结果如下：

```
>>>
=================RESTART: C:\Python\example6.11.py================
12
24
>>>
```

说明：

(1) 函数 add_three()中,x 为必选参数,y 和 z 都为默认值参数。

(2) 第一次调用函数时,只给必选参数 x 传递数值 3,y 和 z 都使用默认值。第二次调用函数时,按照位置参数的对应关系,将数值 7 传递给必选参数 x,同时给默认值参数 y 和 z 也传递新值 8 和 9,从而改变了其原来的默认值。

(3) 需要强调的是,定义函数时必须保证必选参数在前,默认值参数在后。

(4) 默认值参数必须指向不可变对象,可变对象不能作为默认值参数的值,避免造成程序错误,如不能将列表作为参数的默认值。

6.3.4 可变参数

前面定义的函数必须预先定义形参的个数,但在实际应用中,有时并不能事先确定函数到底需要多少个参数,或者参数的数量可以根据调用时的具体情况有所变化,此时就不应该将形参的个数固定。为了解决这一问题,在 Python 中,可以定义可变参数,不事先指定参数的数量,调用函数时,可变参数可以接收任意多个参数。

1. 单星号参数——接收元组

可以通过带星号(*)的参数定义接收可变数量的参数,可变参数在函数调用时自动组装为一个元组。

【例 6.12】 可变参数调用。

程序代码如下:

```
#example6.12
def square_sum(*number):
    print(type(number))
    print("可变参数 number 的值为:",number)
    sum=0
    for i in number:
        sum=sum+i*i
    print("平方和为:",sum)
    return sum
square_sum(1,2,3)
square_sum(2,3,4,5)
square_sum()
```

程序运行结果如下:

```
>>>===============RESTART:C:\Python\example6.12.py===============
<class 'tuple'>
可变参数 number 的值为: (1, 2, 3)
平方和为: 14
<class 'tuple'>
可变参数 number 的值为: (2, 3, 4, 5)
```

```
平方和为：54
<class 'tuple'>
可变参数 number 的值为：()
平方和为：0
>>>
```

说明：

(1) 函数 square_sum()的形参 number 的前面带有星号(*)，说明 number 为可变参数，被调用时可以接收任意多个参数。

(2) 第一次调用时，将 3 个数值传递给参数 number，number 在函数内被组装为包含 3 个元素的元组。

(3) 第二次调用时，将 4 个数值传递给参数 number，number 在函数内被组装为包含 4 个元素的元组。

(4) 第三次调用时，将 0 个参数传递给 number，number 在函数内被组装为包含 0 个元素的元组。

2. 双星号参数——接收字典

定义函数时，通过在参数前面添加两个星号(**)，指定调用时关键字参数被放置在一个字典中传递给函数。如果一个函数定义中的最后一个形参有双星号(**)前缀，所有正常形参之外的其他关键字参数都将被放置在一个字典中传递给该参数。

【例 6.13】 双星号参数。

程序代码如下：

```
#example6.13
def school(x,**y):
    print(type(y))
    print("y:",y)
    print("x:",x)
    for i in y:
        print(i+":"+str(y[i]))
school("985 高校",a="大连理工大学",b="东北大学")
print("-----------------------------------------------------")
school("211 高校",a="大连理工大学",b="东北大学",c="辽宁大学",d="大连海事大学")
print("-----------------------------------------------------")
school("普通省属高校",a="沈阳师范大学",b="渤海大学",c="沈阳工业大学",d="…")
```

```
print("-----------------------------------------------------------")
school("'双一流建设'高校")
```

程序运行结果如下：

```
>>>
=================RESTART:C:\Python\example6.13.py================
<class 'dict'>
y: {'a': '大连理工大学', 'b': '东北大学'}
x: 985 高校
a:大连理工大学
b:东北大学
-----------------------------------------------------------
<class 'dict'>
y: {'a': '大连理工大学', 'b': '东北大学', 'c': '辽宁大学', 'd': '大连海事大学'}
x: 211 高校
a:大连理工大学
b:东北大学
c:辽宁大学
d:大连海事大学
-----------------------------------------------------------
<class 'dict'>
y: {'a': '沈阳师范大学', 'b': '渤海大学', 'c': '沈阳工业大学', 'd': '…'}
x: 普通省属高校
a:沈阳师范大学
b:渤海大学
c:沈阳工业大学
d:…
-----------------------------------------------------------
<class 'dict'>
y: {}
x: '双一流建设'高校
>>>
```

说明：

(1) 函数 school()中定义的参数 y,前面带有两个星号(* *),说明 y 为可变参数,可以接收任意多个参数,且 y 将接收到的参数组装为一个字典。

(2) 第一次调用时,将第一个参数“985 高校”传递给位置参数 x,其他 2 个关键字参

数被组装为一个字典,传递给参数 y。

(3) 第二次调用时,将第一个参数“211 高校”传递给位置参数 x,其他 4 个关键字参数被组装为一个字典,传递给参数 y。

(4) 第三次调用时,将第一个参数“普通省属高校”传递给位置参数 x,其他 4 个关键字参数被组装为一个字典,传递给参数 y。

(5) 第四次调用时,将实参“'双一流建设'高校”传递给必选参数 x,可选参数 y 接收到零个参数,为空字典。

由于定义函数时采用了可变参数(形参前加了 * 或 * *),在函数调用时除了位置参数之外的其他参数都将自动组装为一个元组或字典传递给形参,此时有一种特殊情况,即实参本身就是一个序列(元组、列表、字典、集合),参数又该如何传递呢? 下面的例子将说明这一问题。

【例 6.14】 形参为序列类型的可变参数传递。

程序代码如下:

```
#example6.14
def square_sum(*number):
    print(number)
    sum=0
    for i in number:
        sum=sum+i*i
    return sum
nums=[1,2,3]
print(square_sum(nums))
```

运行该程序,会产生如下错误:

```
>>>
=================RESTART:C:\Python\example6.14.py===============
([1, 2, 3],)
Traceback (most recent call last):
  File "C:\Python\example6.14.py", line 9, in<module>
    print(square_sum(nums))
  File "C:\Python\example6.14.py", line 6, in square_sum
    sum=sum+i * i
TypeError: can't multiply sequence by non-int of type 'list'
>>>
```

说明：

(1) 形参 number 前面带有星号，为可变参数。

(2) 实参 nums 为列表[1,2,3]，调用函数时会将 nums 参数整体组装为一个元组传递给形参 number，故输出 number 的值为元组([1,2,3])。

(3) 程序继续执行到 sum=sum+i * i，即 sum=0+[1,2,3] * [1,2,3]，语句出错。

其实，对于上面的例子，可以采用最常规的方法调用 square_sum()函数，实现方法如下。

【例 6.15】 修改例 6.14 为正确的调用形式。

程序代码如下：

```
#example6.15
def square_sum(*number):
    print(number)
    sum=0
    for i in number:
        sum=sum+i*i
    return sum
nums=[1,2,3]
print(square_sum(nums[0],nums[1],nums[2]))
```

程序运行结果如下：

```
>>>
=================RESTART:C:/Python/example6.15.py================
(1, 2, 3)
14
>>>
```

说明：调用 square_sum()函数时，传递给可变参数 number 的是三个列表元素的值，在函数内部将这三个值组装为一个元组。

3. 参数传递时的序列解包

例 6.15 的方法虽然能够正确执行，但书写比较繁琐。对于序列类型的实参，可以在前面添加一个星号(*)，将实参的序列进行解包，然后在调用函数时传递给可变参数。

【例 6.16】 将序列解包后传递给可变参数。

程序代码如下：

```
#example6.16
def square_sum(*number):
    print(number)
    sum=0
    for i in number:
        sum=sum+i*i
    return sum
nums=[1,2,3]
print(square_sum(*nums))
```

程序运行结果如下：

```
>>>
================RESTART:C:/Python/example6.16.py================
(1, 2, 3)
14
>>>
```

说明：

(1) 调用函数 square_sum()时，实参 nums 前面添加了星号，将对其进行序列解包。

(2) 将列表 nums 解包为三个数值对象传递给可变参数 number，在函数内部再对这些参数进行组装，形成一个元组，从而程序可以正确执行。

(3) 只要函数中的形参为含有多个单变量的可变参数，而实参又是列表、元组、字典、集合等可迭代对象时，这种在实参前添加星号(*)进行序列解包，然后再传递给形参的方法都可以使用。

【例 6.17】 将实参序列解包后传递给多个普通形参。

程序代码如下：

```
#example6.17
def add_three(x,y,z):
    s=x+y+z
    print(s)
lis=[1,2,3]
add_three(*lis)
tup=(1,2,3)
```

```
add_three(*tup)
dic={1:'x',2:'y',3:'z'}
add_three(*dic)
set_1={1,2,3}
add_three(*set_1)
```

程序运行结果如下：

```
>>>
================RESTART:C:\Python\example6.17.py================
6
6
6
6
>>>
```

说明：

(1) 函数 add_three()包含三个普通参数，注意不是可变参数。

(2) 实参为可迭代对象(元组、列表、字典、集合)，可将实参解包后进行参数传递。

(3) 若不将实参进行序列解包而直接传递，实参与形参个数不相等，程序将会出错。

6.3.5 函数的返回值

返回值在前面的例子中已经有所应用，指函数返回给主调用程序的结果，通过函数中的关键字 return 实现，return 后面的值就是函数的返回值。格式如下：

```
return<表达式列表>
```

return 后面可以返回多个表达式的值，执行完 return 语句后函数结束。

【例 6.18】 包含单个 return 语句的函数。

程序代码如下：

```
#example6.18
def cal(x,y):
    s=x+y
    a=x*y
```

```
    p=x**y
    return s,a,p
print(cal(2,3))
```

程序运行结果如下：

```
>>>
=================RESTART:C:/Python/example6.18.py================
(5, 6, 8)
>>>
```

说明：

(1) 函数中的 return 语句返回三个变量的值。

(2) 函数内部没有输出语句，调用函数时必须用 print 语句才能输出函数的返回值，故调用函数的语句不能写成 cal(2,3)。

【例 6.19】 包含多个 return 语句的函数。

程序代码如下：

```
#example6.19
def min(x,y):
    if x<y:
        return x
    else:
        return y
print(min(13,25))
print(min(98,12))
```

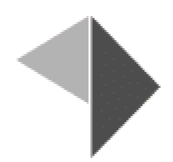
程序运行结果如下：

```
>>>
=================RESTART:C:/Python/example6.19.py================
13
12
>>>
```

说明：

(1) 一个函数可以有多条 return 语句，但执行到第一条 return 语句时函数就结束。

(2) 一个函数也可以没有 return 语句,此时函数没有返回值,或者使用 return None。



6.4 变量的作用域

变量的作用域指变量的作用范围,作用域不相同,即使变量名相同,变量之间也不相互影响。根据变量作用域的不同,可以将变量分为全局变量(Global variables)和局部变量(Local variables)。全局变量的作用域是整个程序,这种变量在整个程序范围内都可以被引用;局部变量定义在函数体内部,它的作用域只在函数内,只能在函数内被引用,一旦程序执行离开函数,变量将失效,不可引用。

在有些程序中,可能会有嵌套函数,即在函数体内部又定义了下一层函数,无论变量被定义在哪一层函数体内,其作用范围都在定义该变量的函数体内部,这些都是局部变量。只有作用域在整个程序范围内的变量才是全局变量。需要注意的是,在 Python 中,非函数和类中写的变量都是全局变量。具体而言,Python 中区分全局变量和局部变量的规则如下。

6.4.1 全局变量

(1) 在主程序中定义的变量是全局变量,确切地说,在主程序中出现在赋值语句等号左侧的变量是全局变量。

(2) 用 global 声明的变量是全局变量,global 语句可以声明多个变量为全局变量,变量中间用逗号隔开。

(3) 没有用 global 语句声明,但出现在函数内赋值号右侧且不是函数参数的变量是全局变量,或者说,如果在函数体内只是引用了某个变量的值而没有为其赋新值,则该变量为全局变量。

6.4.2 局部变量

(1) 函数的形式参数是局部变量。

(2) 在函数体内部赋值号左侧出现的变量是局部变量,作用域在该函数体内部,或者说只要在函数体内部有为变量赋值的操作,该变量即为局部变量。

【例 6.20】 变量的作用域实例 1。

程序代码如下:

```
#example6.20
a,b=3,6                     #a、b定义在主程序中,为全局变量
def func_scope():
    c=a*b                   #c出现在函数体内赋值语句等号左侧,是局部变量
    d=c*2                   #d也是局部变量
    print(c,d)              #在函数体内部,输出局部变量 c、d的值
func_scope()                #在主程序中调用函数
print(a,b)                  #在主程序中输出全局变量 a、b的值
```

程序运行结果如下：

```
>>>
=================RESTART:C:/Python/example6.20.py================
18 36
3 6
>>>
```

说明：

(1) 在函数 func_scope()中,变量 c 和 d 没有被 global 声明为全局变量,且 c 和 d 都出现在了赋值号的左侧,它们都是局部变量,只能在函数 func_scope()内部被引用。

(2) 变量 a 和 b 出现在主程序中的赋值号左侧,虽然没有用 global 语句声明,但仍然为全局变量,主程序调用 func_scope()函数后,执行 print 语句输出变量 a 和 b 的值,仍然为全局变量原来的值。

下面对上面的例子稍作改动,看变量的作用域是否发生变化。

【例 6.21】 变量的作用域实例 2。

程序代码如下：

```
#example6.21
a,b=3,6                    #a、b定义在主程序中,为全局变量
def func_scope():
    a=10                   #a定义在函数体内部的赋值语句等号左侧,为局部变量
    b=20                   #b也是局部变量,函数体内部的 a、b不同于主程序中的 a、b
    c=a*b                  #c出现在函数体内赋值语句等号左侧,是局部变量
    print(a,b,c)
func_scope()
print(a,b)                 #输出全局变量 a、b的值
```

程序运行结果如下：

```
>>>
=================RESTART:C:/Python/example6.21.py================
10 20 200
3 6
>>>
```

说明：

(1) 主程序中的变量 a、b 为全局变量。

(2) 函数 func_scope()内部的变量 a、b 虽然与主程序中的变量 a、b 同名，但由于函数中的 a、b 出现在了赋值语句等号的左侧，说明这两个变量都是局部变量，作用范围只局限在 func_scope()函数内部，与主程序中的 a、b 不是同一变量。

再看下面的例子，进一步厘清局部变量和全局变量的区别。

【例 6.22】 变量的作用域实例 3。

程序代码如下：

```
#example6.22
a,b=3,6                  #a、b定义在主程序中，为全局变量
def func_scope():
    global a             #声明变量 a 为全局变量
    a=10                 #这里的变量 a 与主程序中的变量 a 为同一变量
    b=20                 #b定义在函数体内部的赋值语句等号左侧，为局部变量
    c=a*b                #c出现在函数体内赋值语句等号左侧，为局部变量
    print(a,b,c)
func_scope()
print(a,b)               #输出全局变量 a、b 的值
```

程序运行结果如下：

```
>>>
=================RESTART:C:/Python/example6.22.py================
10 20 200
10 6
>>>
```

说明：

(1) 主程序中定义的变量 a、b 为全局变量。

(2) 函数体内部将变量 a 用 global 语句声明为全局变量,其作用范围为整个程序,因此函数体内部的变量 a 与主程序中的变量 a 为同一变量。

(3) 在函数体内部通过赋值语句改变了全局变量 a 的值,因此在函数体外的主程序中,执行输出语句 print(a,b)时,变量 a 的值发生了变化(由 3 变为 10)。

(4) 函数体内的变量 b 仍为局部变量,与主程序中的变量 b 不是同一变量,故主程序中的输出语句 print(a,b)执行时,b 的值仍为原来的初值 6。



6.5 函数的嵌套

6.5.1 函数的嵌套定义

Python 支持嵌套函数,即在函数定义时,函数体内部又包含另外一个函数的完整定义,并且可以多层嵌套。相对而言,被定义在其他函数体内部的函数称为内部函数,内部函数所在的函数称为外部函数。

【例 6.23】 嵌套函数实例 1。

程序代码如下:

```
#example6.23
def First():                     #定义函数 First()
    a=3
    def Second():                #在函数 First()内部定义函数 Second()
        b=4
        print(a+b)
    Second()                     #在 First()函数内调用 Second()函数
    print(a)
First()                          #在主程序内,调用函数 First()
```

程序运行结果如下:

```
>>>
================RESTART:C:/Python/example6.23.py===============
7
3
>>>
```

说明:

(1) 在函数嵌套定义时,局部变量的作用域为该函数体内部,包括该函数的子函数,

在外部函数中定义的局部变量,相对于内部函数具有隐含的全局变量的意义。

(2) 在外部函数 First()中定义的变量 a 是局部变量,但其作用范围也包含内部函数 Second(),因此在 Second()中可以调用 print()函数直接引用该变量,此时,在 Second()函数看来,变量 a 相当于全局变量。

但是,如果在内部函数中变量 a 出现在了赋值号的左侧,则情况又不一样了,如例 6.24。

【例 6.24】 嵌套函数实例 2。

程序代码如下:

```
#example6.24
def First():                          #定义函数 First()
    a=3
    def Second():                     #在函数 First()内部定义函数 Second()
        a=5
        b=4
        print(a+b)
    Second()                          #在 First()函数内调用 Second()函数
    print(a)
First()
```

程序运行结果如下:

```
>>>
=================RESTART:C:/Python/example6.24.py================
9
3
>>>
```

说明:

(1) 相对于 Second()函数而言,First()内部定义的变量 a 是全局变量,其作用范围包含 Second()函数内部。

(2) 在内部函数 Second()中,变量 a 出现在了赋值号的左侧,所以该变量为函数 Second()重新定义的局部变量,与外部函数 First()中定义的变量 a 为不同变量。

定义多层嵌套函数时,变量的作用域将变得更为复杂,为加深理解,请参考例 6.25。

【例 6.25】 函数多层嵌套实例。

程序代码如下:

```
#example6.25
def first():                    #定义函数 first()
    x1="Dream1"                 #定义函数 first()内的局部变量 x1
    print(x1)
    def second():               #在函数 first()内部定义函数 second()
        x1="second_Dream1"          #定义函数 second()内局部变量 x1,不同于 first()内的 x1
        global x2               #声明 x2 为全局变量
        x2="Dream2"
        print(x1,x2)
        def third():            #在函数 second()内部定义函数 third()
            x3="Dream3"         #定义函数 third()内的局部变量 x3
            print(x1,x2,x3)
        return third()        #second()的返回值为函数 third(),相当于调用 third()
    second()                  #调用函数 second()
    print(x1,x2)              #返回到 first()函数,x1 为前面定义的局部变量
first()                       #调用函数 first()
```

程序运行结果如下：

```
>>>
==================RESTART:C:/Python/example6.25.py================
Dream1
second_Dream1 Dream2
second_Dream1 Dream2 Dream3
Dream1 Dream2
>>>
```

说明：

(1) 函数 first()内的变量 x1 是局部变量,作用范围为整个函数 first()内部,即一直到函数结束调用 first()语句之前,在函数 second()内部的变量 x1 出现在了赋值号的左侧,所以其为 second()函数内的局部变量,与上一层函数 first()内的 x1 不是同一变量。

(2) 在函数 second()内的变量 x2 用 global 语句声明,为全局变量,作用范围为整个程序。

(3) 需要进一步说明的是,如果在函数 third()内部再定义一个变量 x2,即使用一条赋值语句给 x2 赋值,这里的 x2 也是函数 third()内部的局部变量,读者可以尝试修改上面的程序,查看函数输出结果的变化。

6.5.2 lambda 函数

lambda 函数又被称为匿名函数，代表临时使用的简单小函数。

lambda 函数语法格式如下：

```
<函数名>=lambda<参数列表>:<表达式>
```

参数列表即为 lambda 函数包含的形式参数列表，通过对表达式的计算，将函数的返回结果赋值给函数名，因此，lambda 函数相当于下面的普通函数定义格式：

```
def<函数名>(参数列表):
    return<表达式>
```

lambda 函数中的表达式只能为单个表达式，不允许包含复杂语句，但在表达式中可以包含函数，并支持默认值参数和关键字参数，表达式的计算结果相当于普通函数的返回值。下面的示例演示了 lambda 函数的用法：

```
>>>f=lambda x,y,z:x+y+z
>>>f(10,20,30)
60
>>>lm=lambda x,y=3,z=5:x+y+z
>>>lm(4)
12
>>>lm(6,z=5,y=4)
15
```

6.6 递归

由前述可知，在一个函数内部，可以调用其他函数。而在函数内部也可以调用函数自身，这种函数的调用方式被称为递归(recursion)。利用递归，可以把一个大型且复杂的问题层层转化为一个与原问题相似的规模较小的问题求解。递归方法只需少量的程序代码就可描述出解题过程所需要的多次重复计算，大幅度减少了程序的代码量。

递归的核心思想是把原问题分解成规模更小的、具有与原问题有着相同解法的问题。但也并不是所有类似的问题都适合用递归方法解决，应用递归函数解决实际问题，必须满足以下两个条件：

(1) 可以通过递归调用缩小问题规模,但新问题必须与原问题有相同的形式;

(2) 必须存在一种使递归终止的条件,当达到该条件时结束递归。

常用的二分查找算法就是不断地把问题的规模变小(变成原问题的一半),而新问题与原问题有着相同的解法。再比如求 n 的阶乘问题、斐波拉契数列问题等,都可以利用递归的方法实现,下面就利用两个非常典型的实例进一步分析递归的原理,理解函数的递归调用过程。

求 $n!$运算,通常的方法是通过循环语句实现。利用函数的递归调用,同样可以实现求解,而且代码会更简单。

【例 6.26】 利用递归函数求 $n!$。

分析:首先,$n!=n*(n-1)*(n-2)*\cdots*3*2*1$,可以看作为 $n!=n*(n-1)!$,由此,将原来求 $n!$的问题规模变小,变为求$(n-1)!$的问题。同理,继续递推,$(n-1)!=(n-1)*(n-2)!$,进一步缩小问题的规模,变为求$(n-2)!$的问题,因此,$n!=n*(n-1)*(n-2)!$;依此类推,该过程一直进行下去,直到问题的规模最后变为求 2!和 1!的问题。具体过程如下所示:

```
n!=n*(n-1)!
    =n*(n-1)*(n-2)!
    …
    =n*(n-1)*(n-2)*…*3*2!
    =n*(n-1)*(n-2)*…*3*2*1!
```

根据阶乘的定义,1!=1,这是本题递归的终止条件,此时,递归结束,根据 1!=1,再反过来依次求得 2!、3!,最终求得 $n!$。

根据以上分析,求 $n!$满足递归函数解决问题的两个条件:

(1) 要解决的问题:求 $n!$,问题的规模逐步缩小:$n!\rightarrow(n-1)!\rightarrow(n-2)!\cdots2!\rightarrow1!$。

(2) 存在递归的终止条件:当 $n=1$ 时,1!=1。

因此,可以利用函数的递归调用实现求 $n!$,代码如下:

```
#example6.26
def Fact(n):
    if n==1:
        return 1
    else:
        return n*Fact(n-1)
print(Fact(5))
print(Fact(3))
```

执行程序，计算 5 的阶乘和 3 的阶乘，输出结果如下：

```
>>>
=================RESTART:C:/Python/example6.26.py================
120
6
>>>
```

说明：

(1) 本例中，函数 Fact(n)的功能是计算 n!，因为 $n!=n*(n-1)*(n-2)*\cdots*3*2*1$，所以可以将函数 Fact($n$)按如下形式简化：

Fact(n)(=)n * Fact(n-1)

Fact(n)=n * (n-1) * Fact(n-2)

Fact(n)=n * (n-1) * (n-2) * Fact(n-3)

…

Fact(n)=n * (n-1) * (n-2) * … * 3 * Fact(2)

Fact(n)=n * (n-1) * (n-2) * … * 3 * 2 * Fact(1)

Fact(n)=n * (n-1) * (n-2) * … * 3 * 2 * 1

(2) 在这一计算过程中，每次都需要调用同一个函数 Fact()。欲求 n!，需要先利用 Fact()函数求(n-1)!；欲求(n-1)!，需要再次利用 Fact()函数求(n-2)!，以此递推，直到最后求 1!，而 1!=1。在以上过程中，只要 n>1，每次计算都需要调用函数 Fact()自身求(n-1)!。在程序语句中，这一过程都可以用 n * Fact(n-1)表示，函数 Fact()的参数由 n、n-1、…、3、2、1 逐渐减小，直到 n=1 时，不再调用函数 Fact()，给出函数返回值 1。

从上面的例题分析可知：递归的过程分为两个阶段："递"和"归"，"递"即递推，在递推阶段，把较复杂的问题(规模为 n)的求解推到比原问题简单一些的问题(规模小于 n)，函数自上而下不断调用自身，参数依次变化，最终到达递归终止条件；"归"即回归，在回归阶段，从递归终止条件开始，获得问题最简单情况的解，然后逐级返回，函数按照递推相反的顺序自下而上逐层返回函数值，依次得到稍复杂问题的解，最终求得原始问题的解。上面例题中的递归函数 Fact()，当 n=5 时，递归执行 Fact(5)的过程如图 6.1 所示。

"递"的过程如下。

第一层递推：n=5，调用函数 Fact()，Fact(5)=5 * Fact(4)。

第二层递推：n=4，调用函数 Fact()，Fact(4)=4 * Fact(3)。

第三层递推：n=3，调用函数 Fact()，Fact(3)=3 * Fact(2)。

第四层递推：n=2，调用函数 Fact()，Fact(2)=2 * Fact(1)。

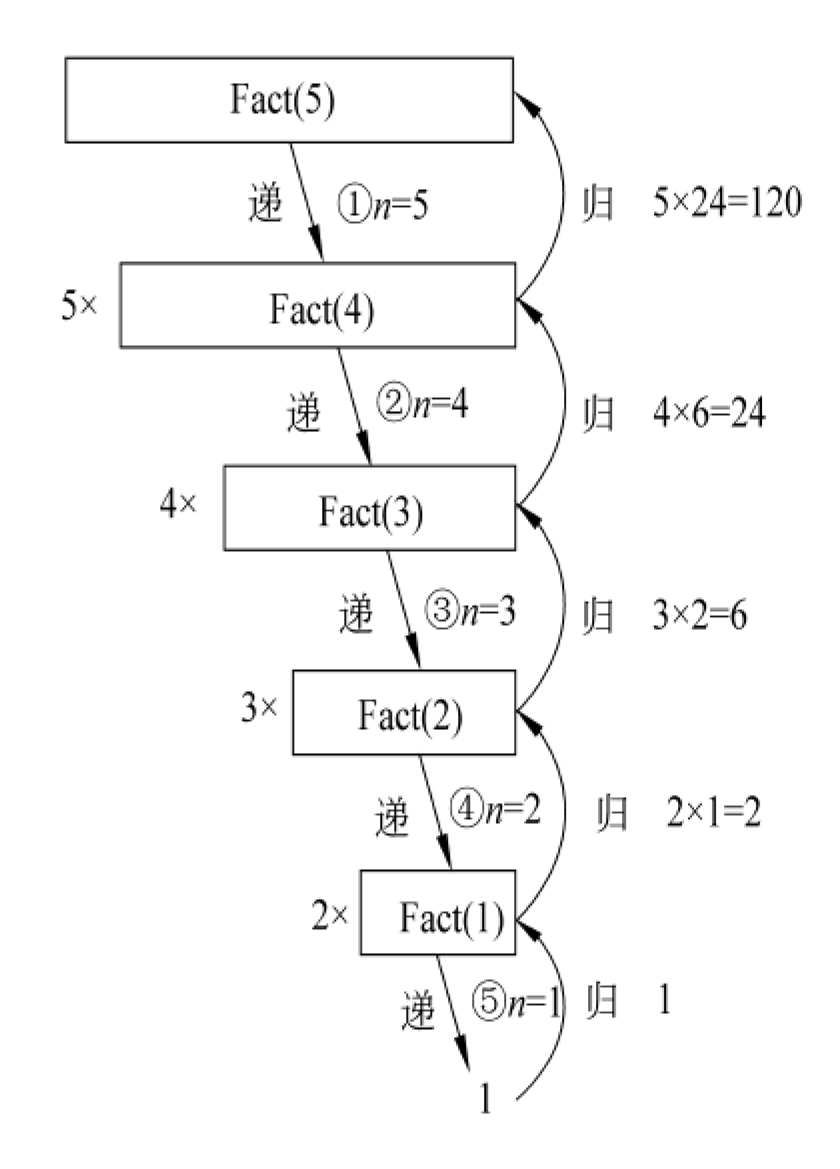


图 6.1　求 Fact(5)的递归过程

第五层递推：n=1，调用函数 Fact()，Fact(1)=1。

此时到达递归的终止条件，得到明确的函数返回值，进入递归调用的第二阶段。

“归”的过程如下。

第一层回归：Fact(1)=1，返回 1。

第二层回归：Fact(2)=2 * Fact(1)=2 * 1=2，返回 2。

第三层回归：Fact(3)=3 * Fact(2)=3 * 2=6，返回 6。

第四层回归：Fact(4)=4 * Fact(3)=4 * 6=24，返回 24。

第五层回归：Fact(5)=5 * Fact(4)=5 * 24=120，返回 120。

此时回到递归调用的第一层，返回原始问题 Fact(5)的解，程序运行结束。

【例 6.27】　利用递归函数求斐波拉契数列(Fibonacci sequence)的第 n 项。

斐波拉契数列以中世纪意大利数学家列昂纳多·斐波拉契(Leonardoda Fibonacci)的名字命名，又称黄金分割数列、费氏数列，指这样一个数列：0、1、1、2、3、5、8、13、21、…，数列的第 0 项是 0，第 1 项是 1，第 2 项是 1，从第 2 项开始，以后的每一项都是其前两项的和。

分析：当 n 大于等于 2 时，求第 n 项斐波拉契数列可以归结为求第 $n-1$ 项和第 $n-2$ 项的斐波拉契数，将原问题规模由 n 降到 $n-1$ 和 $n-2$，同理，求第 $n-1$ 项可以归结为求第 $n-2$ 项和第 $n-3$ 项，以此递推，实现问题规模的缩小且新问题与原问题相同，都为求斐波拉契数列问题，当 n=2 时，第 2 项的值为第 0 项和第 1 项数的和，第 0 项和第 1 项的值分别为 0 和 1，此时递归终止。因此，该过程满足递归函数解决问题的两个条件。

(1) 要解决的问题：求斐波拉契数列的第 n 项。问题的规模逐步缩小为第 n 项→第(n-1)项→第(n-2)项…第 2 项→第 1 项→第 0 项。

(2) 存在递归的终止条件：当 n<2 时，第 1 项的值为 1，第 0 项的值为 0。

程序代码如下：

```
#example6.27
def Fib(n):
    if n<2:
        return n
    else:
        return Fib(n-1)+Fib(n-2)
print(Fib(5))
print(Fib(8))
```

运行程序，输出斐波拉契数列第 5 项和第 8 项的结果如下：

```
>>>
=================RESTART:C:/Python/example6.27.py================
5
21
```

说明：在本例中，求解 Fib(n)的问题，递推到求解 Fib(n-1)和 Fib(n-2)的问题。即求 Fib(n)，必须先求 Fib(n-1)和 Fib(n-2)，而求 Fib(n-1)和 Fib(n-2)，又必须先求 Fib(n-3)和 Fib(n-4)。以此类推，直至计算 Fib(1)和 Fib(0)，而 Fib(1)=1，Fib(0)=0，这是递归的终止条件。在回归阶段，当获得最简单情况的 Fib(1)和 Fib(0)的解后，再逐级返回，依次得到稍复杂问题的解，由 Fib(1)和 Fib(0)的解，返回得到 Fib(2)的结果，由 Fib(2)和 Fib(1)的解，得到 Fib(3)的结果……在得到了 Fib(n-1)和 Fib(n-2)的结果后，最后返回得到 Fib(n)的结果。求 Fib(5)的过程如图 6.2 所示。

采用递归调用的优势是代码简洁、结构清晰。但递归的使用也有它的劣势，因为递归要进行多层函数调用，所以会消耗很多堆栈空间和函数调用时间，还有可能出现堆栈溢出的情况。从上面 Fib(5)的调用过程可以看出，Fib(1)调用了 5 次，Fib(2)调用了 3 次，Fib(3)调用了 2 次，随着规模的增加，函数调用的次数会进一步增加，这个算法的复杂度呈指数级，因此递归调用的程序往往执行效率比较低。

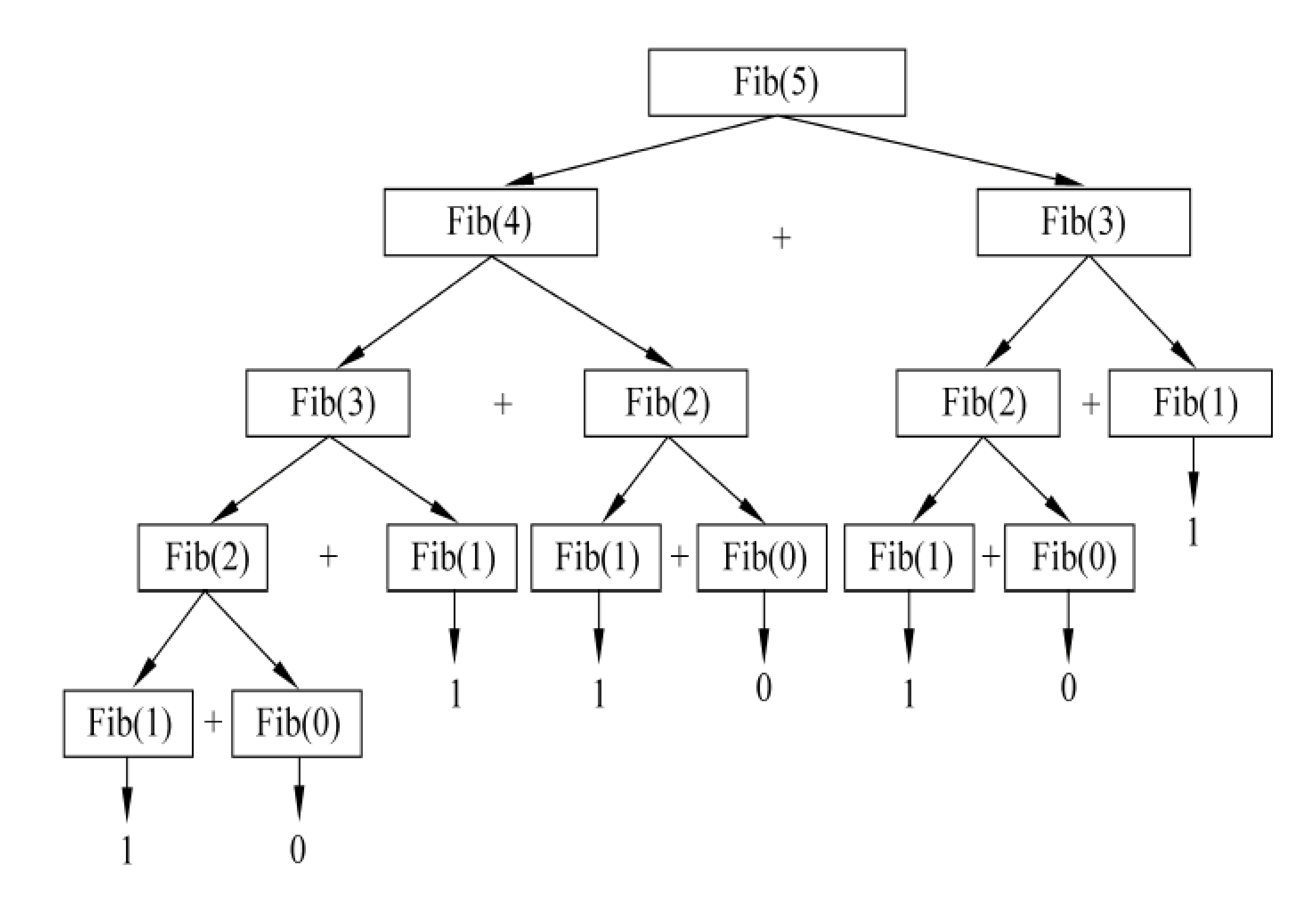
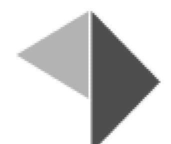


图 6.2　Fib(5)的递归过程



6.7　模块的使用

模块(module)是用来组织 Python 程序代码的一种方法。在前面的几个章节中基本上是用 Python 解释器编程,当从 Python 解释器退出后再次进入时,原先定义的所有方法和变量就都消失了,原代码不能重用,程序的效率很低。并且,随着程序开发过程中代码越写越多,在一个文件中的代码也会越来越长,越来越难以维护。为了解决以上两个方面的问题,提高代码的重用性和可维护性,Python 提供了一个方法,可以把原来的代码存放在一个文件中,也可以把很多函数分组,分别放到不同的文件中,这样,每个文件包含的代码就相对较少,容易维护。这个文件被称为模块(module)。在 Python 中,一个 py 文件被称为一个模块(module)。模块是一个包含所有定义的函数和变量的文件,其后缀名是 py。模块可以被别的程序引入,以使用该模块中的函数等功能,这也是使用 Python 标准模块的方法。

6.7.1　模块的导入

默认情况下,Python 仅安装基本模块,基本模块内的对象可以直接使用。在需要时可以加载其他需要的标准模块和扩展模块。在 Python 中用关键字 import 引入某个模块,具体有三种不同的 import 语句格式。

1. 导入整个模块

```
import<模块名>[as<别名>]
```

当解释器遇到 import 语句时，就会在搜索路径搜索指定的模块。搜索路径是一个特定目录的集合，Python 系统只在这些特定的路径中搜索模块文件名。搜索路径是在 Python 编译或安装时确定的，如果模块的文件名在当前的搜索路径存在就会被导入。默认的搜索路径被存储在 sys 模块的 path 变量中，在交互式解释器中，可以输入以下代码查看搜索路径。

```
>>>import sys
>>>sys.path
['', 'C:\Users\syc\AppData\Local\Programs\Python\Python36\Lib\idlelib', 'C:\Users\syc\
AppData\Local\Programs\Python\Python36\python36.zip', 'C:\Users\syc\AppData\Local\
Programs\Python\Python36\DLLs', 'C:\Users\syc\AppData\Local\Programs\Python\Python36\
lib', 'C:\Users\syc\AppData\Local\Programs\Python\Python36', 'C:\Users\syc\AppData\
Local\Programs\Python\Python36\lib\site-packages']
```

sys. path 语句输出了一个列表，其中第一项是空串('')，代表当前目录，就是执行 Python 解释器的目录。也可以通过调用列表的 append()方法向 sys. path 中增加目录，如：

```
>>>import sys
>>>sys.path.append('要增加的目录路径')
```

Python 本身内置了很多模块，这些模块可以利用 import 语句直接导入。如上面的代码中导入了模块 sys，在文件最开始的地方用 import sys 导入。使用这种方式导入模块后，使用模块内的函数(方法)时必须以“模块名. 函数名”的方式进行访问，如调用 sys 模块中的 path 函数时，可以用 sys. path。别名是可选项，给模块指定一个别名，引用对象时就可以使用“别名. 函数名”的方式了。

```
>>>import math
>>>math.sqrt(9)              #sqrt()函数返回数字的平方根
3.0
>>>import numpy as np        #导入 numpy 模块并设置别名为 np,需要先安装 numpy 模块
```

```
>>>a=np.array((0,1,2,3,4,5))    #通过模块的别名访问其中的对象
>>>print(a)
[0 1 2 3 4 5]
```

如果只用到模块中的一个函数(或者可能用到模块中的所有函数),此时只需明确地单独导入该函数(或者导入该模块下的所有函数)即可。

2. 导入模块下的单个函数

语句格式如下:

```
from<模块名>import<函数名>[as<别名>]
```

3. 导入模块下的所有函数

语句格式如下:

```
from<模块名>import *
```

如果通过这种方式导入,明确指出导入模块中的具体函数而不导入整个模块,调用函数时只需给出函数名,不能再写模块名,如:

```
>>>from math import sin
>>>sin(45)                 #sin()返回弧度的正弦值,不能写成 math.sin(45)
0.8509035245341184
>>>from math import sqrt
>>>sqrt(9)                 #不能写成 math.sqrt(9)
3.0
>>>from math import *
>>>sin(60)
-0.3048106211022167
>>>sqrt(25)
5.0
```

6.7.2　自定义模块和包

1. 自定义模块

在 Python 中,每个 Python 文件就是一个模块,文件的名字就是模块的名字。因此,

可以创建 Python 文件(扩展名为 py),作为模块被导入并使用模块内部的函数(方法)。

下面定义了一个文件 maths.py,在 maths.py 中定义了函数 add():

```
#maths.py
def add(a,b):
    return a+b
```

在 Python 交互环境下或其他程序中就可以调用 maths.py 模块并使用其中的函数 add。如下所示:

```
>>>import maths
>>>maths.add(5,8)
13
```

任何一个 Python 文件既可以在 Python 交互环境下独立运行,也可以作为模块被导入。通过__name__可以识别 Python 文件的使用方式,如果 Python 文件在交互环境下独立运行,则 Python 解释器把__name__属性置为__main__;如果 Python 文件作为模块被导入,则 Python 解释器把__name__属性设置为模块名(不包含扩展名)。利用这一功能可以让一个模块通过交互环境运行时执行一些额外的代码,或者在不同的使用方式下执行不同的代码。

【例 6.28】 Python 程序文件的两种使用方式。

程序代码如下:

```
#example6.28
#因为 import 导入的文件名不包含扩展名,即不能有句点,故将程序文件保存为 example628.py
def test_module():
    if __name__=='__main__':
        print("程序独立运行!")
    elif __name__=='example628':#此处不能写成'example628.py'
        print("程序作为模块被调用!")
test_module()
```

直接运行该程序,得到的结果如下:

```
>>>
==================RESTART:C:/Python/example628.py=================
程序独立运行!
```

在 Python 的 IDLE 交互模式下使用 import 语句将该程序作为模块导入时,得到的结果如下(注意:程序文件必须保存在 sys.path 包含的任意一个目录下,否则无法导入):

```
>>>import example628
程序作为模块被调用!
>>>
```

2. 模块的组织——包

为了避免不同人编写的模块名产生冲突,Python 引入了按目录组织模块的方法,称为包(Package)。包可以看作包含大量 Python 程序模块的文件夹,在包的每个目录中都必须包含一个__init__.py 文件,用来声明当前文件夹是一个包,__init__.py 可以是空文件,也可以有 Python 代码。如果没有这个文件,Python 就把这个目录当成普通目录,而不是一个包。例如,已有一个名字为 test 的模块文件 test.py,如果再定义一个 test 模块,就会与原模块产生冲突,此时可以通过包组织模块,避免冲突。方法是选择一个顶层包名,如 myprogram,按照如下目录存放:

```
myprogram/
__init__.py
        test.py
```

引入了包的组织形式后,引用模块的名字就需要在模块名前面加上包的名字。现在,引用 test.py 模块的名字就变成了 mycompany.test。只要顶层的包名不产生冲突,不同包内的模块即使重名也不会产生冲突。在一个包内可以包含多个模块,类似地,在一个包内也可以包含多个包,组成多级目录,形成多级层次的包结构。例如,可能有类似下面结构的包,顶层包 sounds 内包含 3 个子包 formats、effects 和 filters,在每个子包内都用文件__init__.py 声明该目录是一个包而不是普通文件夹,在每个子包内又包含多个 py 文件,每个 py 文件都是一个模块,不同包内的模块可以重名,不会产生冲突。

```
sounds/                        顶层包 sounds
    __init__.py                声明 sounds 目录是一个包
    formats/                   formats 子包
            __init__.py        声明 formats 目录是一个包
            wavread.py
            wavwrite.py
```

```
            aiffread.py
            aiffwrite.py
            auread.py
            auwrite.py
            ...
    effects/                                #effects 子包
            __init__.py                     #声明 effects 目录是一个包
            echo.py
            surround.py
            reverse.py
            ...
    filters/                                #filters 子包
            __init__.py                     #声明 filters 目录是一个包
            equalizer.py
            vocoder.py
            karaoke.py
            ...
```

6.7.3 安装第三方模块

Python 之所以流行的一个重要原因是其支持数量众多、涉及各领域开发、功能强大的第三方模块(扩展库)。安装第三方模块有多种不同的方法和工具,其中,采用包管理工具 pip 是目前的主流方式。采用 pip 方式,首先要求计算机必须联网,通过简单命令即可实现对第三方模块的安装和卸载等操作。常用的 pip 命令的使用方式如表 6.1 所示。

表 6.1 常用 pip 命令使用方法

pip 命令示例	说　明
piplist	列出当前已安装的所有模块
pipinstall	安装模块
pipuninstall	卸载模块
pipinstall-upgrade	升级模块
pip download	下载模块
pipshow	显示模块信息
pipsearch	查找模块

一般来说，第三方模块都会在 Python 官方的 pypi. python. org 网站注册，想要安装一个第三方模块，可以先在 pypi 上搜索。截至本书编写完成时，该网站已经收录了 105711 个涉及各领域的第三方模块，并且还在以每天几十个的速度增加。

下面以安装第三方模块 Pillow 为例，介绍利用 pip 命令安装第三方模块的过程。Pillow 是 Python 图像处理的第三方模块，包含改变图像大小、旋转图像、图像格式转换、色场空间转换、图像增强等基本图像处理功能的函数(方法)。

首先，使用 pip 命令安装 Python 第三方模块必须在命令提示符环境中进行，并且要切换到 pip 命令所在的目录。然后，按照执行如下命令安装 Pillow 第三方模块：

```
:\>pipinstall Pillow
```

该命令执行后会通过网络下载 Pillow 并安装，执行过程如图 6.3 所示。

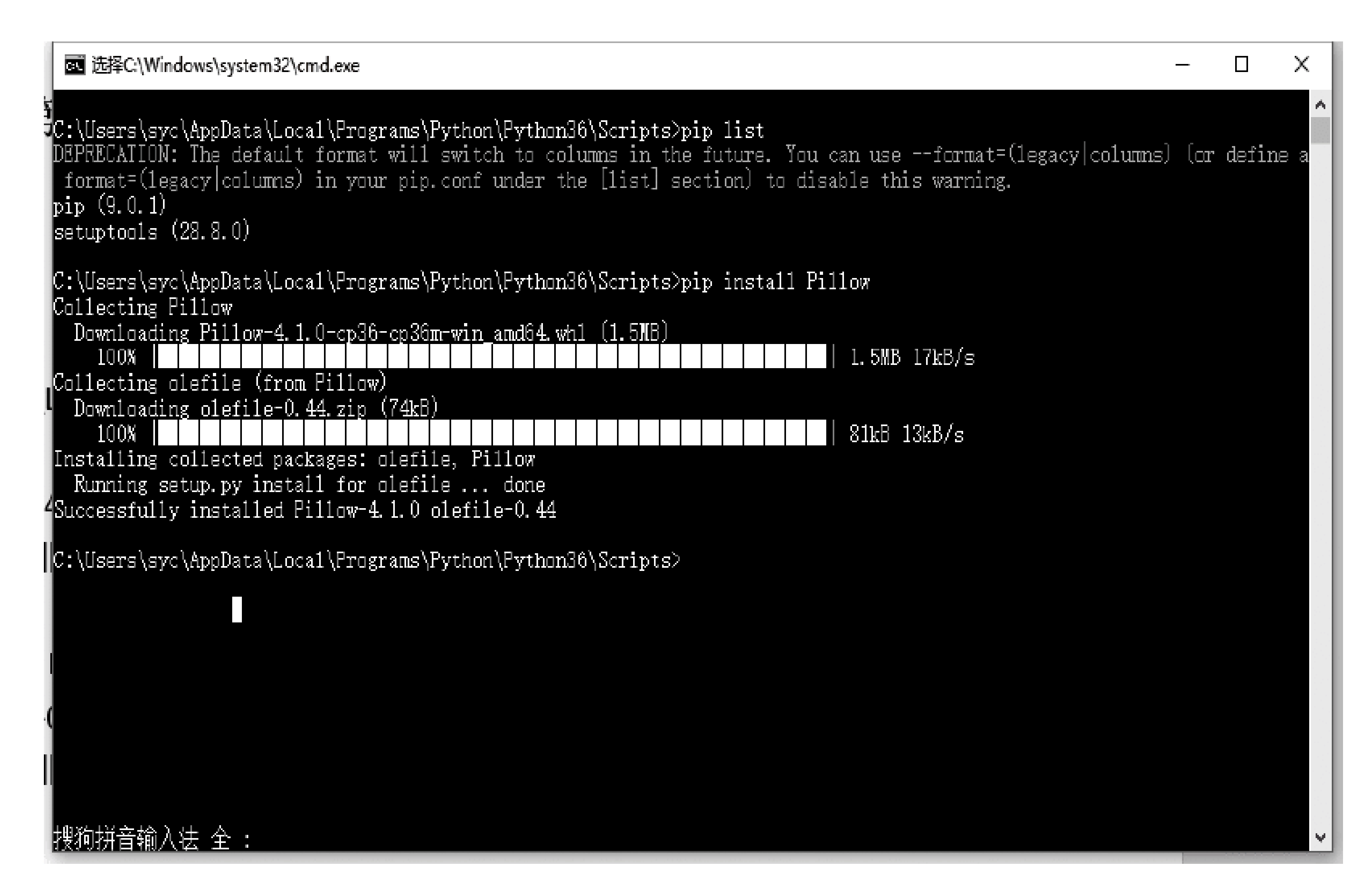


图 6.3　用 pip 命令安装 Pillow 模块

Pillow 第三方模块安装成功后，就可以在 Python 开发环境中导入该模块并使用其中的函数。例如，安装完 Pillow 后，可以执行类似下面的代码使用它：

```
>>>from PIL import Image               #导入第三方模块 Pillow的 Image类
>>>im=Image.open('flower.jpg')         #打开图片文件 flower.jpg
>>>print(im.size)                      #输出图片的分辨率大小(宽,高)
(1836, 3264)
```

受限于网络等因素，直接使用 pip 安装有时会非常慢，且容易失败。此时可以使用镜

像安装或者利用 setuptools 等包管理工具,这方面的内容读者可以查阅相关资料,本书不进行介绍。

卸载一个模块使用 pip uninstall 命令,如卸载 Pillow 模块,如:

```
:\>pip uninstall Pillow
```

也可以通过 piplist 命令查看系统中已经安装的所有第三方模块,如:

```
:\>pip list
beautifulsoup4(4.5.3)
olefile(0.44)
Pillow(4.1.0)
pip(9.0.1)
requests(2.13.0)
setuptools(28.8.0)
...
```

使用模块可以大幅度提高代码的可维护性,同时,编写代码不必从零开始。当一个模块编写完毕,就可以在其他场合被引用。编写模块时,也不必考虑变量或函数的名字是否会与其他模块冲突,相同名字的函数和变量完全可以分别存储在不同的模块中。

6.7.4 常见模块应用实例

1. 图形绘制模块 turtle

Python 内置的标准模块 turtle 可以非常方便地进行图形绘制,turtle 原意为海龟,turtle 的绘图过程就是模拟一只海龟在 Python 坐标系中爬行,可以有前进、后退、旋转等基本爬行动作,其爬行经过的轨迹就是绘制的图形。初始状态下,海龟从坐标原点(0,0)即画布正中央开始水平向右爬行,类似于数学上的直角坐标系,向右为 x 轴正方向,向上为 y 轴正方向。在利用 turtle 进行绘制图形的过程中,需要用到 turtle 模块中的各种函数,下面就介绍一下其中最重要的两大类函数。更多的函数请读者参考 Python 官方文档(https://docs.python.org/2/library/turtle.html)中关于 turtle 的介绍。

(1) 画笔控制函数

常用画笔控制函数及功能见表 6.2。

(2) 形状绘制函数

常用形状绘制函数及功能见表 6.3。

表 6.2　turtle 常用画笔控制函数

函　数　名	功　　能
pendown()或者 pd()或者 down()	按下画笔,之后海龟的移动将绘制形状
penup()或者 pu()或者 up()	抬起画笔,之后海龟的移动将不绘制形状
pensize()或者 width()	设置画笔尺寸
color()	设置画笔颜色和填充颜色
pencolor()	设置画笔颜色
fillcolor()	设置填充颜色

表 6.3　turtle 常用形状绘制函数

函　数　名	功　　能
forward(distance)或者 fd(distance)	向海龟当前行进方向前进 distance 距离
backward(distance)或者 bk(distance)或者 back(distance)	向海龟当前行进方向后退 distance 距离
right(angle)或者 rt(angle)	海龟向右转 rangle 角度
left(angle)或者 lt(angle)	海龟向左转 rangle 角度
goto(x,y=None)或者 setpos(x,y=None)或者 setposition(x,y=None)	海龟移动到绝对坐标位置
setheading(to_angle)或者 seth(to_angle)	海龟当前行进方向为 to_angle 绝对方向角度: 0°——正东向(海龟初始爬行方向) 90°——正北向(上方向) 180°——正西向(左方向) 270°——正南向(下方向)
home()	海龟回到起始坐标原点(0,0)
circle(radius, extent=None)	省略 extent,绘制圆形,半径为 radius 给定 extent,绘制弧形,弧形角度为 extent
speed((speed=None)	设置海龟爬行的速度,速度为 0~10 之间的整数

使用 turtle 模块,还有一个常用的函数 setup(),往往在代码初始阶段使用,用来设置绘制图形的主窗口大小及位置。使用 setup()函数的格式如下:

```
setup(width, height, startx, starty)
```

各参数含义如下。

width：窗口宽度(整数为像素,小数为窗口宽度与屏幕的比例)。

height：窗口高度(整数为像素,小数为窗口宽度与屏幕的比例)。

startx：窗口左侧与屏幕左侧的像素距离,如果是 None,则窗口位于屏幕水平中央。

starty：窗口顶部与屏幕顶部的像素距离,如果是 None,则窗口位于屏幕垂直中央。

【例 6.29】 利用 turtle 模块绘制多边形。

程序代码如下：

```
#example6.29
import turtle
turtle.setup(680,280,200,200)
turtle.pensize(5)
turtle.speed(3)
turtle.color("red")
turtle.up()
def polygon(x,y,keep,fd,angel):
    turtle.goto(x,y)
    turtle.down()
    for i in range(keep):
        turtle.forward(fd)
        turtle.right(angel)
    turtle.up()
polygon(-300,100,4,180,360/4)          #绘制正方形
polygon(-80,100,3,200,360/3)           #绘制三角形
polygon(160,100,6,100,360/6)           #绘制正六边形
```

程序运行结果如图 6.4 所示。

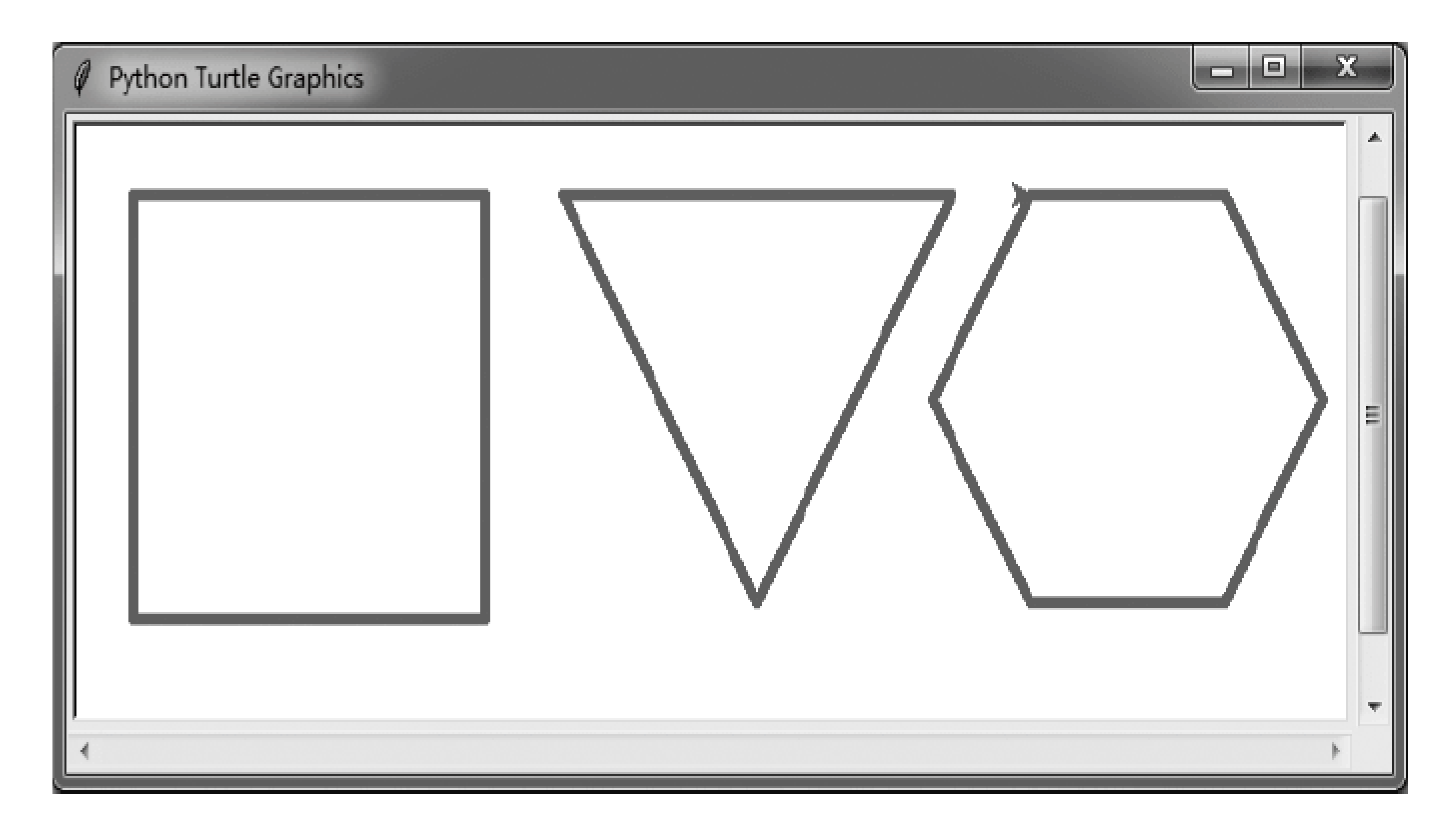


图 6.4　例 6.29 的运行结果

说明：

(1) 绘制多边形的功能封装在自定义函数 polygon()中。

(2) 调用自定义函数的坐标参数可以根据窗口大小及绘图位置调整。

【例 6.30】 利用 turtle 模块绘制圆形及曲线。

程序代码如下：

```
#example6.30
import turtle
def circles(x,y,rad,ang=None):
    turtle.up()
    turtle.goto(x,y)
    turtle.down()
    turtle.circle(rad,ang)
    turtle.up()
def littlesnake(x,y,loop,rad,ang):
    turtle.goto(x,y)
    turtle.down()
    for i in range(loop):
        turtle.circle(rad,ang)
        turtle.circle(-rad,ang)
    turtle.up()
turtle.setup(680,300,200,200)
turtle.pensize(10)
turtle.speed(3)
turtle.color("blue")
circles(x=-220,y=-100,rad=100)                        #绘制圆形
circles(x=-100,y=-100,rad=100,ang=180)                #绘制左半圆
circles(x=100,y=-100,rad=-100,ang=180)                #绘制右半圆
littlesnake(x=160,y=-100,loop=4,rad=20,ang=80)  #绘制小蛇曲线
```

程序运行结果如图 6.5 所示。

说明：

(1) circle()函数的第二个参数为角度，当不设置角度参数时，将绘制圆形，设置角度参数可以绘制半圆、四分之一圆等各种弧形。

(2) circle()函数的第一个参数为半径，当半径为正数时，按照圆心在运动轨迹的左侧(逆时针方向)绘制，当半径为负数时，按照圆心在运动轨迹的右侧(顺时针方向)绘制。

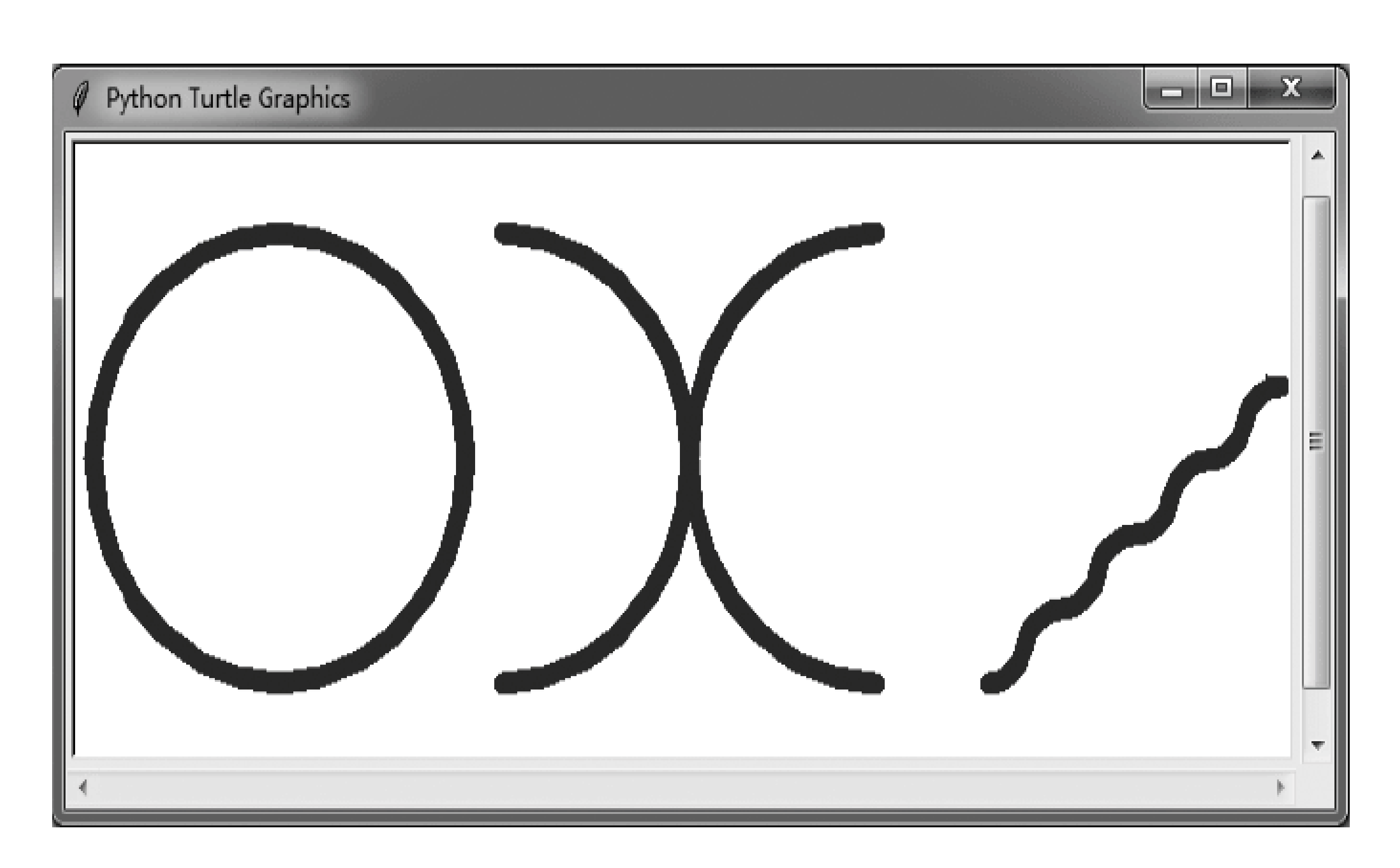


图 6.5　例 6.30 的运行结果

2. 中文分词模块 jieba

中文分词(Chinese Word Segmentation)指将一个汉字序列切分成一个个单独的词。众所周知,在英文的行文中,单词之间是以空格作为自然分界符的,而中文的词没有形式上的分界符,因此中文分词比英文要复杂得多、困难得多。jieba 是 Python 中一个支持中文分词的第三方模块,具有强大的中文分词功能。作为第三方模块,首先应该使用 pip 命令安装 jieba,然后才能使用,具体命令如下:

```
:\>pip install jieba
```

jieba 模块支持 3 种分词模式。

(1) 精确模式:将句子最精确地切开,适合文本分析。

(2) 全模式:把句子中所有可以成词的词语都扫描出来,速度非常快,但是不能解决歧义。

(3) 搜索引擎模式:在精确模式的基础上,对长词再次切分,提高召回率,适合用于搜索引擎分词。

除此之外,jieba 还支持繁体字分词和自定义词典。jieba 包含的主要函数如表 6.4 所示。

上面几个函数主要实现分词以及向词典中添加新词的功能,除了这些之外,jieba 模块还有一些函数支持关键词提取、词性标注等功能,在此不做深入介绍,单纯从分词的角度,这几个函数已经足够用了。下面通过几个语句演示以上函数的功能。

表 6.4 jieba 主要函数

函 数 名	功 能
jieba.cut(s)	精确模式，返回可迭代类型 generator
jieba.lcuts(s)	精确模式，返回列表类型
jieba.cut(s,cut_all=True)	全模式，输出文本 s 中所有可能的值，返回可迭代类型 generator
jieba.lcut(s,cut_all=True)	全模式，返回列表类型
jieba.cut_for_search(s)	搜索引擎模式，适合搜索引擎建立索引的分词结果，粒度比较细
jieba.lcut_for_search(s)	搜索引擎模式，返回一个列表类型
jieba.add_word(w)	向分词词典中增加新词 w

精确模式：

```
>>>word=jieba.cut("今天天气真好,我们一起去远足吧!")
>>>list(word)
['今天天气', '真', '好', ',', '我们', '一起', '去', '远足', '吧', '!']
>>>jieba.lcut("今天天气真好,我们一起去远足吧!")
['今天天气', '真', '好', ',', '我们', '一起', '去', '远足', '吧', '!']
>>>
```

全模式：

```
>>>word=jieba.cut("今天天气真好,我们一起去远足吧!",cut_all=True)
>>>list(word)
['今天', '今天天气', '天天', '天气', '真好', '', '', '我们', '一起', '去', '远足',
'吧', '', '']
>>>jieba.lcut("今天天气真好,我们一起去远足吧!",cut_all=True)
['今天', '今天天气', '天天', '天气', '真好', '', '', '我们', '一起', '去', '远足',
'吧', '', '']
>>>
```

搜索引擎模式：

```
>>>word=jieba.cut_for_search("今天天气真好,我们一起去远足吧!")
>>>list(word)
['今天', '天天', '天气', '今天天气', '真', '好', ',', '我们', '一起', '去', '远足', '吧',
'!']
```

```
>>>jieba.lcut_for_search("今天天气真好,我们一起去远足吧!")
['今天', '天天', '天气', '今天天气', '真', '好', ',', '我们', '一起', '去', '远足', '吧',
'!']
>>>
```

向词典中增加新词:

```
>>>jieba.lcut("你段誉到底是真的不会,还是装傻?")
['你', '段', '誉', '到底', '是', '真的', '不会', ',', '还是', '装傻', '?']
>>>jieba.add_word("段誉")                #段誉为《天龙八部》中的人物
>>>jieba.lcut("你段誉到底是真的不会,还是装傻?")
['你', '段誉', '到底', '是', '真的', '不会', ',', '还是', '装傻', '?']
```

说明:

(1) 无论是哪种模式,cut()函数与 lcut()函数分词的结果是相同的,区别是返回值的类型不同,由于 lcut()函数返回的列表类型比较灵活,因此经常被使用。

(2) 精确模式的结果是完整的,而且没有多余;全模式返回的结果包含所有可能的中文词语,比较全面,但冗余也最大;搜索引擎模式首先执行精确模式,然后再对结果中的长词进一步切分。

【例 6.31】 统计《天龙八部》中出场次数排名前 10 位的人物。

程序代码如下:

```
#example6.30
import jieba
txt=open("天龙八部.txt", "r", encoding='utf-8').read()
words=jieba.lcut(txt)                        #精确分词,返回一个列表
counts={}                                    #用字典类型存储人物及出现的次数
for word in words:
    if len(word)!=1:                         #不统计单个汉字的词汇
        counts[word]=counts.get(word,0)+1
listitem=list(counts.items())                #将字典转换为列表
#按列表中每个元组第二个位置元素的值(即人数)降序排序
#key指明排序前被调用的函数,reverse指定降序排序
listitem.sort(key=lambda x:x[1], reverse=True)
print("{0:<10}{1:>8}".format("人物","出场次数"))
for i in range(10):                          #输出排序后的列表前 10 项
    word, count=listitem[i]
    print ("{0:-<10}{1:->10}".format(word, count))
```

程序运行结果如下：

```
>>>
=================RESTART:C:\Python\example6.30.py================
Building prefix dict from the default dictionary …
Loading model from cache C:\Users\syc\AppData\Local\Temp\jieba.cache
Loading model cost 1.184 seconds.
Prefix dict has been built succesfully.
人物出场次数
说道--------------2016
段誉--------------1881
自己--------------1575
虚竹--------------1467
一个--------------1380
什么--------------1265
萧峰--------------1231
不是--------------1050
武功--------------1020
甚么---------------997
>>>
```

说明：

(1) 实际上，这段程序实现的功能是统计文件中所有词汇出现的次数，但是真正想要输出的是人物出现的次数。

(2) 输出结果中包含多个与人名无关的词汇，如“自己”“一个”“什么”等，应将这些词汇排除，根据结果中的这些词汇，可以建立排除词库，经过多次运行程序，不断发现结果中那些与人名无关的词汇，逐步将它们添加到排除词库中，经过多次修改，最后的程序代码如例 6.32 所示。

【例 6.32】 例 6.31 改进版。

程序代码如下：

```
#example6.31
import jieba
excludes={"说道","自己","一个","什么","不是","武功",
          "甚么","一声","咱们","不知","师父","心中",
          "知道","出来","如何","姑娘","便是","突然",
          "如此","他们","之中","只见","不能","大理",
```

```
                "丐帮","只是","不敢","弟子"}        #排除词库
txt=open("天龙八部.txt", "r", encoding='utf-8').read()
words=jieba.lcut(txt)
counts={}                                          #用字典类型存储人物及出现的次数
for word in words:
    if len(word)!=1:                               #单个汉字的词汇不做统计
        counts[word]=counts.get(word,0)+1
for word in excludes:
    del(counts[word])                          #删除与人名无关的词汇
listitem=list(counts.items())                  #因为字典类型不能排序,将字典转换为列表
#按列表中每个元组第二个位置元素的值(即人数)降序排序
#key指明排序前被调用的函数,reverse指定降序排序
listitem.sort(key=lambda x:x[1], reverse=True)
print("{0:<10}{1:>8}".format("人物","出场次数"))
for i in range(10):                            #输出排序后的列表前10项
    word, count=listitem[i]
    print ("{0:-<10}{1:->10}".format(word, count))
```

程序运行结果如下：

```
>>>
================RESTART:C:\Python\example6.31.py================
Building prefix dict from the default dictionary …
Loading model from cache C:\Users\syc\AppData\Local\Temp\jieba.cache
Loading model cost 1.189 seconds.
Prefix dict has been built succesfully.
人物出场次数
段誉--------------1881
虚竹--------------1467
萧峰--------------1231
乔峰---------------832
王语嫣--------------814
慕容复--------------781
段正淳--------------706
木婉清--------------688
鸠摩智--------------528
游坦之--------------508
>>>
```

说明：

(1) 观察得到的结果，萧峰与乔峰为小说中的同一个人物，应将这两组数据整合到一组。

(2) 小说中有两个非常重要的人物“阿朱”和“阿紫”并没有在结果中出现，似乎不合常理，这可能是由于中文语义的复杂性造成的，同时，为了增加人物统计的正确性，可以将以上统计出的人物都添加到 jieba 的词典中。

修改后的程序如例 6.33 所示。

【例 6.33】 例 6.32 的改进版。

程序代码如下：

```
#example6.32
import jieba
excludes={"说道","自己","一个","什么","不是","武功",
          "甚么","一声","咱们","不知","师父","心中",
          "知道","出来","如何","姑娘","便是","突然",
          "如此","他们","之中","只见","不能","大理",
          "丐帮","只是","不敢","弟子","见到","声音",
          "众人","内力","我们","南海","心想","倘若"}
txt=open("天龙八部.txt", "r", encoding='utf-8').read()
list1=["段誉","虚竹","萧峰","乔峰","王语嫣","慕容复",
       "段正淳","木婉清","鸠摩智","游坦之","阿朱","阿紫"]
for i in list1:                    #将人物添加到 jieba 词典
    jieba.add_word(i)
words=jieba.lcut(txt)
counts={}                          #用字典类型存储人物及出现的次数
for word in words:
    if len(word)==1:               #不统计单个汉字的词汇
        continue
    elif word=="萧峰":
        rword="乔峰"
    else:
        rword=word
    counts[rword]=counts.get(rword,0)+1
for word in excludes:
    del(counts[word])
listitem=list(counts.items())      #将字典转换为列表
```

```
#按列表中每个元组第二个位置元素的值(即人数)降序排序
#key指明排序前被调用的函数,reverse指定降序排序
listitem.sort(key=lambda x:x[1], reverse=True)
print("{0:<10}{1:>8}".format("人物","出场次数"))
for i in range(10):            #输出排序后的列表前10项
    word, count=listitem[i]
    print ("{0:-<10}{1:->10}".format(word, count))
```

程序运行结果如下：

```
>>>
=================RESTART:C:\Python\example6.32.py================
Building prefix dict from the default dictionary …
Loading model from cache C:\Users\syc\AppData\Local\Temp\jieba.cache
Loading model cost 1.209 seconds.
Prefix dict has been built succesfully.
人物出场次数
段誉--------------2380
乔峰--------------2130
虚竹--------------1467
阿紫---------------898
阿朱---------------882
王语嫣--------------814
慕容复--------------781
段正淳--------------706
木婉清--------------688
鸠摩智--------------528
>>>
```

3. 词云模块 wordcloud

词云也称为文字云、标签云等,是指对文本中出现频率较高的关键词予以视觉上的突出,形成关键词云层或关键词渲染。利用 Python 的 jieba 模块和 wordcloud 模块可以轻松实现词云的构建。

wordcloud 模块最重要的子模块是 WordCloud,创建词云最常见的功能都可以通过 WordCloud 子模块实现。WordCloud 包含很多参数,具体如下：

```
wordcloud.WordCloud(font_path=None, width=400, height=200, margin=2, ranks_only=None,
prefer_horizontal=0.9, mask=None, scale=1, color_func=None, max_words=200, min_font_
size=4, stopwords=None, random_state=None, background_color='black', max_font_size=
None, font_step=1, mode='RGB', relative_scaling=0.5, regexp=None, collocations=True,
colormap=None, normalize_plurals=True)
```

WordCloud 常见参数的含义如表 6.5 所示，其他参数的含义可参考官方网站 http://amueller.github.io/word_cloud/generated/wordcloud.WordCloud.html#wordcloud.WordCloud。

表 6.5 WordCloud 常用参数

参数名	功能
font_path	使用的字体路径，支持 otf 或 ttf 格式文件，默认字体是 DroidSansMono，显示中文词云必须设置该参数
width	画布的宽度，整数类型
height	画布的高度，整数类型
prefer_horizontal	词语水平方向排版出现的频率，默认为 0.9，词语垂直方向排版出现频率默认为 0.1
mask	如果为空，则使用二维遮罩绘制词云；如果 mask 非空，设置的宽高值将被忽略，遮罩形状被 mask 取代，全白的部分将不会绘制，其余部分会用于绘制词云
scale	按照比例进行放大画布，如设置为 1.5，则长和宽都是原来画布的 1.5 倍
max_words	显示词语的最大数量，默认值为 200
min_font_size	显示的最小字号，默认为 4 号
max_font_size	显示的最大字号
stopwords	屏蔽词，如果为空，则使用内置的 STOPWORDS
background_color	词云的背景色
color_func	生成新颜色的函数，如果为空，则使用 self.color_func
colormap	给每个单词随机分配颜色，若指定 color_func，则忽略该方法

WordCloud 的常用函数如表 6.6 所示。

表 6.6 WordCloud 常用函数

函　　数	功　　能
fit_words(frequencies)	依据词语和频率生成词云 frequencies 包含了词语和频率的数组
generate(text)或者 generate_from_text(text)	根据文本生成词云
generate_from_frequencies(frequencies[…])	根据词语和频率生成词云,同 fit_words(frequencies)
recolor([random_state, color_func, colormap])	对现有输出重新着色,速度比重新生成整个词云快。random_state:如果非空,会使用固定的随机状态;如果给出一个整数,它会被用作 random.Random 状态的一个种子 color_func:生成新颜色的函数,如果为空,则使用 self.color_func
to_file(filename)	输出到文件

WordCloud 是第三方模块,需要单独安装。在 Windows 系统下,有时直接使用 pip install wordcloud 命令安装会出错,可以从官方网站下载 whl 文件,然后利用 pip 命令安装 whl 文件。如本书下载的 whl 文件为 wordcloud-1.3.1-cp36-cp36m-win_amd64.whl,在命令提示符下,可以用下面的命令安装 wordcloud 模块:

```
:\>pip install wordcloud-1.3.1-cp36-cp36m-win_amd64.whl
```

安装后,就可以使用 wordcloud 模块制作词云了。

【例 6.34】 《天龙八部》词云统计。

程序代码如下:

```
#example6.34
import jieba
import matplotlib.pyplot as plt
from wordcloud import WordCloud
txt=open("天龙八部.txt", "r", encoding='utf-8').read()
list1=["段誉","虚竹","萧峰","乔峰","王语嫣","慕容复",
          "段正淳","木婉清","鸠摩智","游坦之","阿朱","阿紫"]
for i in list1:                    #将主要人物添加到 jieba 词典
    jieba.add_word(i)
words=jieba.lcut(txt)
word_split=" ".join(words)         #以空格为分隔符,连接分词后的所有词汇
my_wordclud=WordCloud(max_words=100,width=2600,height=1600,\
```

```
                    background_color='white',\
                  font_path='msyhbd.ttf').generate(word_split)#微软雅黑
plt.imshow(my_wordclud)             #在坐标区域显示绘制的词云图像
plt.axis("off")                     #不显示坐标轴标签
plt.show()                          #显示创建的绘图对象
```

程序运行结果如图 6.6 所示。



图 6.6　例 6.34 的运行结果

说明：

(1) 为保证对人物统计的准确性，将小说中的主要人物添加到 jieba 词库。

(2) Matplotlib 库是 Python 中用于科学计算的数据可视化第三方库，其中的 pyplot 子库提供了和 MATLAB 类似的绘图 API，提供一些基础绘图功能，在本例中用来将生成的词云图像显示出来。



习题 6

一、填空题

1. Python 安装扩展库常用的工具是________。

2. 使用 pip 工具查看当前已安装的 Python 扩展库的完整命令是________。

3. 在函数内部可以通过关键字________定义全局变量。

4. 如果函数中没有 return 语句或者 return 语句不带任何返回值，那么该函数的返回值为________。

5. 已知 $g=\text{lambda}\ x,\ y=3,\ z=5: x*y*z$，则语句 print(g(2)) 的输出结果为________。

6. 在主程序中定义函数如下：

```
def demo(*p):
    s=sum(p)
    return s
```

在主程序中调用该函数，表达式 demo(1，2，3)和表达式 demo(1，2，3，4)的值分别为________和________。

7. 在主程序中定义函数如下：

```
def func(**p):
    s=sum(p.values())
    return(s)
```

在主程序中调用该函数，表达式 func(x=3，y=4，z=5)的值为________。

二、判断题

1. pip 命令也支持扩展名为 whl 的文件直接安装 Python 扩展库。 (   )

2. 调用函数时，在实参前面加一个星号 * 表示序列解包。 (   )

3. 定义函数时，即使该函数不需要接收任何参数，也必须保留一对空的圆括号表示这是一个函数。 (   )

4. 一个函数如果带有默认值参数，那么所有参数必须都设置默认值。 (   )

5. 定义 Python 函数时必须指定函数返回值的类型。 (   )

6. 定义 Python 函数时，如果函数中没有 return 语句，则默认返回空值 None。 (   )

7. 函数中必须包含 return 语句。 (   )

8. 函数中的 return 语句一定能够得到执行。 (   )

9. 不同作用域中的同名变量之间互相不影响，也就是说，在不同的作用域内可以定义同名的变量。 (   )

10. 全局变量会增加不同函数之间的隐式耦合度，从而降低代码可读性，因此应尽量避免过多地使用全局变量。 ( )

11. 在函数内部没有办法定义全局变量。 ( )

12. 函数内部定义的局部变量当函数调用结束后会被自动删除。 ( )

13. 调用带有默认值参数的函数时，不能为默认值参数传递任何值，必须使用函数定义时设置的默认值。 ( )

14. 在同一个作用域内，局部变量会隐藏同名的全局变量。 ( )

15. 形参可以看作是函数内部的局部变量，函数运行结束之后形参就不可访问了。 ( )

16. 在函数内部没有任何声明的情况下直接为某个变量赋值，这个变量一定是函数内部的局部变量。 ( )

17. 在 Python 中定义函数时不需要声明函数参数的类型。 ( )

18. 在函数中没有任何办法可以通过形参影响实参的值。 ( )

19. 在定义函数时，某个参数名字前面带有一个 * 符号表示其是可变长度参数，可以接收任意多个普通实参并存放于一个元组之中。 ( )

20. 在定义函数时，某个参数名字前面带有两个 * 符号表示其是可变长度参数，可以接收任意多个关键参数并将其存放于一个字典之中。 ( )

21. 在定义函数时，带有默认值的参数必须出现在参数列表的最右端，任何一个带有默认值的参数右边不允许出现没有默认值的参数。 ( )

22. 在调用函数时，可以通过关键参数的形式进行传值，从而避免必须记住函数形参顺序的麻烦。 ( )

23. 在调用函数时，必须牢记函数形参的顺序才能正确传值。 ( )

24. 调用函数时传递的实参个数必须与函数形参个数相等。 ( )

25. 执行语句 from math import sin 之后，可以直接使用 sin()函数，例如 sin(5)。 ( )

三、阅读程序

1. 分析下面程序运行的结果。

```
def square_sum(number):
    sum=0
    for i in number:
        sum=sum+i * i
    print(sum)
```

```
square_sum((1,2,3))
square_sum([1,2,3])
square_sum({1,2,3})
```

2. 分析下面程序运行的结果。

```
def square_sum(number):
    sum=0
    for i in number:
        sum=sum+i*i
    print(sum)
m=[1,2,3]
square_sum(*m)
```

3. 分析下面程序运行的结果。

```
def hello_python():
print('hello Python')
def three_hellos():
    for i in range(3):
        hello_python()
three_hellos()
```

四、程序设计

1. 编写一个程序求 $1!+2!+3!+\cdots+n!$,要求编写一个自定义函数用来求 $n!$,然后利用该函数求 $1\sim n$ 的阶乘和。

2. 利用 turtle 模块设计程序,绘制一个五角星。

第7章

Python类和对象

学习目标

- 理解面向对象程序设计的基本概念，掌握类和对象的定义和使用方法
- 掌握类的属性和方法的灵活应用
- 理解类的继承和派生的概念，掌握继承的使用方法
- 理解多态的概念，掌握方法重载和运算符重载的方法



7.1 面向对象编程

7.1.1 面向过程与面向对象

Python 虽然是一种解释语言，但是它既支持面向过程的编程，也支持面向对象的编程，而 Python 从设计之初就已经是一门面向对象的语言。

1. 面向过程

面向过程(Procedure Oriented Programming)的程序设计方法是把求解问题中的数据定义为不同的数据结构，以功能为中心进行设计，用一个函数实现一个功能。面向过程的程序设计方法中，所有的数据都是公用的，一个函数可以使用任何一组数据，而一组数据又能被多个函数所使用，函数与其操作的数据是分离的；其控制流程由程序中预定的顺序决定，通过分析得出解决问题所需要的步骤(功能)，然后用函数把这些步骤一一实现，使用时再依次对函数进行调用。

前面介绍的内容主要体现的是 Python 面向过程的程序设计方法。

2. 面向对象

面向对象(Object Oriented Programming)的程序设计方法是把求解问题中所有的独

立个体都看作各自不同的对象,将数据和对数据的操作方法都封装在一起,数据和操作是一个相互依存、不可分割的整体,这个整体就是对象。将相同类型对象的共性进行抽象就形成了类,而为了类能够与外界进行联系,在类中必须声明一些函数(方法),这些方法用于与外界进行通信。

面向对象是以数据为中心描述系统,其控制流程由运行时各种事件的实际发生触发,而不再由预定顺序决定,它建立对象的目的不是为了完成一个个步骤,而是为了描述某个对象在整个解决问题的步骤中的行为。

7.1.2 面向对象的相关概念

面向对象方法是接近人们日常生活中处理问题思路的新方法,其基本出发点就是尽可能按照人类认识世界的方法和思维方式分析和解决问题。面向对象以对象为最基本的元素,是一种由对象、类、封装、继承和多态等概念构造系统的软件开发方法。

1. 对象

对象(Object)是现实世界中客观存在的某种事物,可以将人们感兴趣或要加以研究的事、物和概念等都称为对象。对象既能表示结构化的数据,也能表示抽象的事件、规则以及复杂的工程实体等。如自然界的交通工具、房屋建筑、山、水、人、动物等,也可以是生活中的一种逻辑结构或抽象概念(如部门、班级或体育比赛等)。

在面向对象的系统中,对象是一个将数据属性和操作行为封装起来的实体。数据描述了对象的状态,是对象的静态特性;操作用来描述对象的动态特性,是行为或功能,可以操纵数据,改变对象的状态。

Python 中对象的概念与其他面向对象的程序设计语言略有不同,其范围更加广泛。Python 中的一切内容都可以是对象,而不一定必须是某个类的实例。如字符串、列表、元组、字典等内置数据类型都具有和类完全相似的语法和用法。

2. 类

类(Class)是人们对客观事物的高度抽象。抽象是指抓住事物的本质特性,找出事物之间的共性,并将具有共同特性的事物划分为一类,得到一个抽象的概念。例如,人、汽车、房屋、水果等都是类的例子。

类是一种类型,是具有相同属性和操作行为的一组对象的集合。类和对象的关系是抽象与具体的关系,类的作用是定义对象,类给出了属于该类的全部对象的抽象定义,而对象则是类的具体化,是符合这种定义的一个类的实例。类还可以有子类和父类,子类通过对父类的继承,形成层次结构。

把一组对象的共同特性加以抽象并存储在一个类中的能力，是面向对象技术中最重要的一点；是否建立了一个丰富的类库，则是衡量一个面向对象程序设计语言成熟与否的重要标志。

3. 封装

封装(Encapsulation)是面向对象方法的重要特征之一，是指将对象的属性和行为(数据和操作)包裹起来形成一个封装体。该封装体内包含对象的属性和行为，对象的属性由若干个数据组成，而对象的行为则由若干操作组成，这些操作是通过函数实现的，也称为方法。

封装体具有独立性和隐藏性。独立性是指封装体内所包含的属性和行为构成了一个不可分割的独立单位。隐藏性是指封装体内的某些成员(数据或者方法)在封装体外是不可见的，既不能被访问，也不能被改变，这部分成员被隐藏了，具有安全性。一般地，封装体和外界的联系是通过接口(函数)进行的。

4. 继承

继承(Inheritance)是面向对象方法的另一个重要特征，是提高重用性的重要措施。继承提供了创建新类的一种方法，表现了特殊类与一般类的关系。特殊类具有一般类的全部属性和行为，并且还具有自己特殊属性和行为，这就是特殊类对一般类的继承。通常将一般类称为基类(父类)，而将特殊类称为派生类(子类)。

使用继承可以使人们对事物的描述变得简单。例如已经描述了动物这个类的属性和行为，由于哺乳动物是动物的一种，它除了具有动物这个类的所有属性和行为外，还具有自己的特殊属性和行为，这样在描述哺乳动物时就只需要在继承动物类的基础上再加入哺乳动物所特有的属性和行为即可。因此哺乳动物是特殊类，是子类，而动物是一般类，是父类。

继承的本质特征就是行为共享，通过行为共享，可以减少冗余性，很好地解决软件重用性的问题。

5. 多态性

多态性(Polymorphism)指一种行为对应多种不同的实现，即对象根据接收的消息而做出动作，同样的消息在被不同的对象接收时可产生完全不同的结果。多态性的表现就是允许不同类的对象对同一消息做出响应，即同一消息可以调用不同的方法，而实现的细节则由接收对象自行决定。



7.2 类的定义与对象的创建

类是一种用户自定义的数据类型，是对具有共同属性和行为的一类事物的抽象描述，共同属性被描述为类中的数据成员，共同行为被描述为类中的成员函数。类是对象的抽象，而对象是类的具体实例，在使用过程中，必须先定义类，然后才能用它定义和使用对象。Python的类具有所有面向对象程序设计语言的标准特征，而且具备Python特有的动态特点，即类在程序运行时创建，生成后也可以修改。

7.2.1 类的定义格式

Python使用关键字class定义类，并在类中定义属性(数据成员)和方法(成员函数)，格式如下：

```
class<类名>:
    <属性定义>
    <方法定义>
```

其中，class为关键字，类名的首字母通常为大写字母。

【例7.1】 类的定义。

程序代码如下：

```
#example7.1
class Person:                                               #声明类 Person
    number=0                                                #类属性
    def __init__(self, name, gender,age):                   #类的构造函数
        self.name=name                                      #初始化对象属性
        self.gender=gender
        self.age=age
        Person.number+=1
    def displayPerson(self):                                #类的方法
        print('Name:',self.name,'Gender:',self.gender,'Age:',self.age)
    def displayNumber(self):                                #类的方法
        print('Total person:',Person.number)
```

说明：

(1) 该例定义了一个类Person，其中number为类属性，name、gender和age为实例

属性，displayPerson()和 displayNumber()为方法。

(2) __init__()是类的特殊方法，称为构造函数，在创建对象时系统自动调用，用于初始化实例属性(对象属性)。

(3) self 是类中定义方法时参数表的一个参数，定义方法时参数列表至少要有一个参数，通常 self 被指定为第一个参数，表示所创建的对象。

(4) 类属性和实例属性是不同的概念，本书将在后面进行详细说明。

7.2.2　对象的创建

对象是类的实例，对象的创建过程也是类的实例化过程。创建对象和调用函数类似，如果构造函数__init__()声明有参数，则还需要传入相应的参数；同时，创建对象后还要把它赋给一个变量，使该变量指向对象，否则将无法引用所创建的对象。

【例 7.2】 对象的创建。

程序代码如下：

```
#example7.2
class Person:
    number=0
    def __init__(self, name, gender,age):
        self.name=name
        self.gender=gender
        self.age=age
        Person.number+=1
    def displayPerson(self):
        print('Name:',self.name,'Gender:',self.gender,'Age:',self.age)
    def displayNumber(self):
        print('Total person:',Person.number)

stu1=Person('Liming','M',19)                #创建对象 stu1 并初始化
stu2=Person('Zhangli','F',20)               #创建对象 stu2 并初始化
stu1.displayPerson()                        #调用方法显示对象 stu1 属性
stu2.displayPerson()                        #调用方法显示对象 stu2 属性
print('Total students:',Person.number)      #输出类属性 number
stu1.displayNumber()                        #调用方法显示类属性 number
stu2.displayNumber()
```

程序运行结果如下：

```
>>>
==============RESTART:C:\Users\python3.6\example7.2.py=============
Name: Liming Gender: M Age: 19
Name: Zhangli Gender: F Age: 20
Total students: 2
Total person: 2
Total person: 2
>>>
```

说明：

(1) 创建对象时需指定相应的参数，在构造函数调用时进行参数的传递。

(2) 调用类的方法需使用“.”操作符指明调用哪个对象的方法，如 stu1.displayPerson()，访问属性也可以通过“.”直接进行，如 stu1.age 和 Person.number。

(3) 显示类属性 number 既可以直接输出，也可以通过调用对象方法 displayNumber() 实现。

在类定义外也可以根据需要通过对象随时添加、修改或删除属性。

【例 7.3】 属性的添加、修改和删除。

程序代码如下：

```
#example7.3
class Person:
    number=0
    def __init__(self, name, gender,age):
        self.name=name
        self.gender=gender
        self.age=age
        Person.number+=1
    def displayPerson(self):
        print('Name:',self.name,'Gender:',self.gender,'Age:',self.age)
    def displayNumber(self):
        print('Total person:',Person.number)

stu1=Person('Liming','M',19)
stu2=Person('Zhangli','F',20)
stu1.score=90                               #添加属性 score
stu2.score=85                               #添加属性 score
stu1.displayPerson()
```

```
stu2.displayPerson()
print('The score of the first student:',stu1.score)
print('The score of the second student:',stu2.score)
stu1.age=21                              #修改属性 age
del stu1.score                           #删除属性 score
stu1.displayPerson()
print('The score of the first student:',stu1.score)
```

程序运行结果如下：

```
>>>
==============RESTART:C:/Users/python3.6/example7.3.py=============
Name: Liming Gender: M Age: 19
Name: Zhangli Gender: F Age: 20
The score of the first student: 90
The score of the second student: 85
Name: Liming Gender: M Age: 21
Traceback (most recent call last):
File "C:/Users/python3.6/example7.3.py", line 25, in<module>
print('The score of the first student:',stu1.score)
AttributeError: 'Person' object has no attribute 'score'
>>>
```

注意，由于对象 stu1 的属性 score 已经删除，因此再次输出信息时会出错。

属性的添加、修改和删除也可以通过调用函数完成，如例 7.4。

【例 7.4】 通过函数调用进行属性的添加、修改和删除。

程序代码如下：

```
#example7.4
class Person:
    number=0
    def __init__(self, name, gender,age):
        self.name=name
        self.gender=gender
        self.age=age
        Person.number+=1
    def displayPerson(self):
        print('Name:',self.name,'Gender:',self.gender,'Age:',self.age)
    def displayNumber(self):
```

```
print('Total person:',Person.number)

stu1=Person('Liming','M',19)
stu2=Person('Zhangli','F',20)
setattr(stu1,'score',90)                  #创建一个新属性 score,并赋值
print(getattr(stu1,'score'))              #返回属性的值
print('The score of the first student:',stu1.score)
delattr(stu1,'score')                     #删除属性 score
print(hasattr(stu1,'score'))              #如果存在属性 score,则返回 True
```

程序运行结果如下：

```
>>>
==============RESTART:C:/Users/python3.6/example7.4.py=============
90
The score of the first student: 90
False
>>>
```

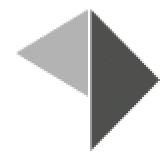
说明：

(1) 函数 getattr(对象,属性名)的功能是访问对象的属性。

(2) 函数 hasattr(对象,属性名)的功能是检查是否存在一个属性,结果为逻辑值。

(3) 函数 setattr(对象,属性名,属性值)的功能是设置一个属性,如果属性不存在,则创建一个新属性。

(4) 函数 delattr(对象,属性名)的功能是删除属性。



7.3 属性和方法

类由属性和方法组成,属性是对数据的封装,方法是对象所具有的行为,但是属性和方法又因其是属于类还是对象而表现出不同的特性;同时,属性和方法又可以分为公有的和私有的。在 Python 语言中,属性和方法的公有和私有是通过标识符的约定区分的。

7.3.1 类属性与对象属性

根据所属的对象,Python 的属性分为类属性和对象属性(也称实例属性)两种。类属性是在类中方法之外定义的属性,既可以通过类名访问,也可以通过对象名访问;而对象

属性只为单独的特定对象拥有,可以在类外显示定义,也可以在类的构造函数__init__()中定义,定义时以 self 作为前缀,且只能通过对象名访问。

【例 7.5】 类属性和对象属性。

程序代码如下:

```
#example7.5
class Person:
    number=0                            #类属性
    def __init__(self, name, gender,age):    #初始化对象属性
        self.name=name
        self.gender=gender
        self.age=age
        Person.number+=1

stu1=Person('Liming','M',19)
stu2=Person('Zhangli','F',20)
print('Name:',stu1.name,'Gender:',stu1.gender,'Age:',stu1.age)
print('Name:',stu2.name,'Gender:',stu2.gender,'Age:',stu2.age)
stu1.score=90                          #对象属性 score 在类外定义
stu2.score=85                          #对象属性 score 在类外定义
print('The score of the first student:',stu1.score)
print('The score of the second student:',stu2.score)
print(stu2.number)
print(Person.number)
print(Person.name)
```

程序运行结果如下:

```
>>>
==============RESTART:C:/Users/python3.6/example7.5.py=============
Name: Liming Gender: M Age: 19
Name: Zhangli Gender: F Age: 20
The score of the first student: 90
The score of the second student: 85
2
2
Traceback (most recent call last):
File "C:/Users/python3.6/example7.5.py", line 20, in<module>
```

```
    print(Person.name)
AttributeError: type object 'Person' has no attribute 'name'
>>>
```

说明：

(1) Person类有一个类属性number，既可以通过类名访问Person.number，也可以通过对象名访问stu2.number。

(2) name、gender和age是对象属性，在构造函数中定义，score也是对象属性，但是在类外定义。

(3) 对象属性只能通过对象名访问，而不能通过类名访问，因此在执行print(Person.name)时出错。

7.3.2 公有属性与私有属性

Python的公有属性和私有属性通过属性命名方式区分，如果属性名以两个下画线开头，则说明是私有属性，否则是公有属性。私有属性的访问通过如下形式进行：

```
<类(对象)名>.<_类名__私有属性名>
```

【例7.6】 公有属性和私有属性。

程序代码如下：

```
#example7.6
class Car:
    salesPrice=150000                 #公有类属性
    __manufacturePrice=120000         #私有类属性
    def __init__(self,brand,serial):
        self.brand=brand              #公有对象属性
        self.__serial=serial          #私有对象属性

print('Public data salesPrice',Car.salesPrice)
print('Private data manufacturePrice',Car._Car__manufacturePrice)
c1=Car('丰田','卡罗拉')
print('Public data brand of c1',c1.brand)
print('Private data serial of c1',c1._Car__serial)
```

程序运行结果如下：

```
>>>
==============RESTART:C:/Users/python3.6/example7.6.py=============
Public data salesPrice 150000
Private data manufacturePrice 120000
Public data brand of c1 丰田
Private data serial of c1 卡罗拉
>>>
```

说明：

(1) Car 类有一个公有类属性 salesPrice，可以通过类名直接访问 Car.salesPrice，私有类属性__manufacturePrice 则需要通过特定方式访问 Car._Car__manufacturePrice。

(2) Car 类的公有对象属性 brand 可以直接通过对象名访问 c1.brand，私有对象属性则需要通过特定方式访问 c1._Car__serial。

7.3.3 对象方法

Python 方法可以分为公有方法和私有方法、类方法以及静态方法。其中公有方法和私有方法都属于对象，公有方法的定义无须特别说明，而私有方法在定义时，方法名要以两个下画线开头。

每个对象都有自己的公有方法和私有方法，在这两类方法中可以访问类和对象的属性，这两类方法也都可以通过类或者对象调用。公有方法通过对象名可直接调用，但是私有方法不能通过对象名直接调用，只能在属于对象的公有方法中通过 self 调用。如果通过类的方式调用方法，则必须传入一个对象，而且在调用私有方法时还要采用如下调用格式：

```
<类(对象)名>.<_类名__私有方法名>()
```

这种格式既可以实现类的方式调用，也可以实现对象方式的调用。

【例 7.7】 公有方法和私有方法。

程序代码如下：

```
#example7.7
class Methods:
    def publicMethod1(self):            #定义公有方法
        print('公有方法 publicMethod!')
    def __privateMethod(self):          #定义私有方法
        print('私有方法 privateMethod!')
    def publicMethod2(self):            #定义公有方法
```

```
        self.__privateMethod()

m=Methods()
m.publicMethod1()                    #通过对象调用公有方法
Methods.publicMethod1(m)             #通过类名调用公有方法
m.publicMethod2()                    #通过对象的公有方法调用私有方法
m._Methods__privateMethod()          #通过对象名调用私有方法
Methods._Methods__privateMethod(m)   #通过类名调用私有方法
```

程序运行结果如下：

```
>>>
==============RESTART:C:/Users/python3.6/example7.7.py=============
公有方法 publicMethod!
公有方法 publicMethod!
私有方法 privateMethod!
私有方法 privateMethod!
私有方法 privateMethod!
>>>
```

说明：

(1) 类 Methods 中定义了两个公有方法 publicMethod1()和 publicMethod2()，公有方法既可以通过对象名调用，也可以通过类名调用，但是通过类名调用必须传入一个对象，如 Methods. publicMethod1(m)。

(2) 类中定义的__privateMethod()为私有方法，也可以通过对象或类名调用，如通过对象的公有方法调用 m. publicMethod2()，通过对象名直接调用 m._Methods__privateMethod()，通过类名调用 Methods._Methods__privateMethod(m)。

(3) 需要注意的是，通过类名调用公有方法和私有方法时都必须传入一个对象，否则会出错。

7.3.4 类方法

类方法属于类，可以通过 Python 的修饰器@classmethod 定义，也可以通过使用内置函数 classmethod()的方式将一个普通的方法转换为类方法。类方法可以通过类名或者对象名访问。

【例 7.8】 类方法。

程序代码如下：

```
#example7.8
class Methods:
    @classmethod                                         #定义公有类方法
    def publicClassMethod(cls):
        print('公有类方法 publicClassMethod!')
    @classmethod                                         #定义私有类方法
    def __privateClassMethod(cls):
        print('私有类方法 privateClassMethod!')
    def publicMethod(self):                              #定义公有方法
        print('普通公有方法 publicMethod!')
    def __privateMethod(self):                           #定义私有方法
        print('普通私有方法 privateMethod!')
    publicMethodToClassMethod=classmethod(publicMethod)          #转换为类方法
    privateMethodToClassMethod=classmethod(__privateMethod)      #转换为类方法

m=Methods()
m.publicClassMethod()
Methods.publicClassMethod()
m._Methods__privateClassMethod()
Methods._Methods__privateClassMethod()
m.publicMethodToClassMethod()
Methods.publicMethodToClassMethod()
m.privateMethodToClassMethod()
Methods.privateMethodToClassMethod()
```

程序运行结果如下：

```
>>>
==============RESTART:C:/Users/python3.6/example7.8.py=============
公有类方法 publicClassMethod!
公有类方法 publicClassMethod!
私有类方法 privateClassMethod!
私有类方法 privateClassMethod!
普通公有方法 publicMethod!
普通公有方法 publicMethod!
```

```
普通私有方法 privateMethod!
普通私有方法 privateMethod!
>>>
```

说明：

(1) 类 Methods 定义了两个类方法，公有的 publicClassMethod 和私有的 privateClassMethod，定义时一般用 cls 作为类方法的第一个参数名称，在调用类方法时不需要为该参数传递值。

(2) 类方法可以通过类名或者对象名访问，如 m. publicClassMethod()和 Methods. publicClassMethod()，m. _Methods __privateClassMethod()和 Methods. _Methods __privateClassMethod()。

(3) 方法 publicMethodToClassMethod 和 privateMethodToClassMethod 是经过转换得到的类方法。

7.3.5 静态方法

静态方法也属于类，可以通过 Python 的修饰器@staticmethod 定义，也同样可以通过使用内置函数 staticmethod()将一个普通的方法转换为静态方法。

【例 7.9】 静态方法。

程序代码如下：

```
#example7.9
class Methods:
    @staticmethod                                    #定义公有静态方法
    def publicStaticMethod():
        print('公有静态方法 publicStaticMethod!')
    @staticmethod                                    #定义私有静态方法
    def __privateStaticMethod():
        print('私有静态方法 privateStaticMethod!')
    def publicMethod(self):                          #定义公有方法
        print('普通公有方法 publicMethod!')
    def __privateMethod(self):                       #定义私有方法
        print('普通私有方法 privateMethod!')
    publicMethodToStaticMethod=staticmethod(publicMethod)          #转换为静态方法
    privateMethodToStaticMethod=staticmethod(__privateMethod)      #转换为静态方法

m=Methods()
```

```
m.publicStaticMethod()
Methods.publicStaticMethod()
m._Methods__privateStaticMethod()
Methods._Methods__privateStaticMethod()
m.publicMethodToStaticMethod(m)
Methods.publicMethodToStaticMethod(m)
m.privateMethodToStaticMethod(m)
Methods.privateMethodToStaticMethod(m)
```

程序运行结果如下：

```
>>>
==============RESTART:C:/Users/python3.6/example7.9.py=============
公有静态方法 publicStaticMethod!
公有静态方法 publicStaticMethod!
私有静态方法 privateStaticMethod!
私有静态方法 privateStaticMethod!
普通公有方法 publicMethod!
普通公有方法 publicMethod!
普通私有方法 privateMethod!
普通私有方法 privateMethod!
>>>
```

说明：

(1) 类 Methods 定义了两个静态方法，定义时无须传入 self 参数或 cls 参数。

(2) 静态方法可以通过类名或者对象名访问，如 m. publicStaticMethod() 和 Methods. publicStaticMethod()，m._Methods__privateStaticMethod() 和 Methods._Methods__privateStaticMethod()。

(3) 方法 publicMethodToStaticMethod() 和 privateMethodToStaticMethod() 是经过转换得到的静态方法，但在调用时必须传入一个对象。

7.3.6　内置方法

Python 类有大量的内置方法，内置方法中的一部分有默认的行为，而另一部分则没有，留到需要的时候实现。这些内置方法是 Python 中用来扩展类的强有力的方式。表 7.1 列出了比较常用的内置方法。

表 7.1 常用的内置方法

内 置 方 法	说　　明
__init__(self,…)	初始化对象,在创建新对象时调用
__del__(self)	释放对象,在对象被删除之前调用
__new__(cls, * args, * * kwd)	实例的生成操作
__str__(self)	在使用 print 语句时被调用
__getitem__(self,key)	获取序列的索引 key 对应的值,等价于 seq[key]
__len__(self)	在调用内联函数 len()时被调用
__cmp__(stc,dst)	比较两个对象 src 和 dst
__getattr__(s,name)	获取属性的值
__setattr__(s,name,value)	设置属性的值
__delattr__(s,name)	删除 name 属性
__getattribute__()	__getattribute__()功能与__getattr__()类似
__gt__(self,other)	判断 self 对象是否大于 other 对象
__lt__(slef,other)	判断 self 对象是否小于 other 对象
__ge__(slef,other)	判断 self 对象是否大于或者等于 other 对象
__le__(slef,other)	判断 self 对象是否小于或者等于 other 对象
__eq__(slef,other)	判断 self 对象是否等于 other 对象
__call__(self, * args)	把实例对象作为函数调用

1. __init__()方法

__init__()方法是 Python 类的一种特殊方法,也称构造函数,当创建对象时系统自动调用,用来为对象分配内存并且为属性进行初始化。用户可以自己设计构造函数,如果没有设计,Python 将提供一个默认的构造函数进行初始化工作。

【例 7.10】 构造函数。

程序代码如下:

```
example7.10
class Person:
    def __init__(self, name, gender,age):
        self.name=name
```

```
        self.gender=gender
        self.age=age

stu1=Person('Liming','M',19)
stu2=Person('Zhangli','F',20)
print('Name:',stu1.name,'Gender:',stu1.gender,'Age:',stu1.age)
print('Name:',stu2.name,'Gender:',stu2.gender,'Age:',stu2.age)
```

程序运行结果如下：

```
>>>
==============RESTART:C:/Users/python3.6/example7.10.py============
Name: Liming Gender: M Age: 19
Name: Zhangli Gender: F Age: 20
>>>
```

说明：该例中的构造函数__init__()用于初始化属性 name、gender 和 age，创建对象时由系统自动调用。

2. __del__()方法

__del__()方法也称析构函数，用来释放对象占用的存储空间，在 Python 删除对象和收回对象存储空间时被自动调用和执行。如果用户没有编写析构函数，则 Python 将提供一个默认的析构函数。

【例 7.11】 析构函数。

程序代码如下：

```
#example7.11
class Person:
    def __init__(self, name, gender,age):
        self.name=name
        self.gender=gender
        self.age=age
    def __del__(self):
        print('调用析构函数：',self.name,self.gender,self.age)

stu1=Person('Liming','M',19)
```

```
stu2=Person('Zhangli','F',20)
print('Name:',stu1.name,'Gender:',stu1.gender,'Age:',stu1.age)
print('Name:',stu2.name,'Gender:',stu2.gender,'Age:',stu2.age)
del stu1
del stu2
```

程序运行结果如下：

```
>>>
==============RESTART:C:/Users/python3.6/example7.11.py============
Name: Liming Gender: M Age: 19
Name: Zhangli Gender: F Age: 20
调用析构函数: Liming M 19
调用析构函数: Zhangli F 20
>>>
```

说明：该例中的析构函数__del__()由用户自己定义，当使用 del 删除对象时，会被系统调用以释放存储空间。



7.4 继承

继承也是面向对象程序设计最重要的特征之一，实现了代码重用，增强了软件模块的可复用性和可扩充性，提高了软件的开发效率。有了继承机制，就可以在已有类的基础上添加新的成员构成新的类，从而提供了定义类的另一种方法。通过继承可以充分利用前人或自己以前的开发成果，同时又能够在开发过程中保持足够的灵活性，不拘泥于复用的模块。Python 提供了类的继承机制，解决了软件重用问题。

7.4.1 继承和派生的概念

继承是由一个已有类创建一个新类的过程。已有类称为基类或父类，新类称为派生类或子类。派生类从基类继承基类的成员，并根据需要添加新的成员，或对原有的成员进行改造（改写），以适应新类的需求。

派生类也可以作为其他类的基类，这个过程可以一直进行下去，从一个基类派生出来的多层类就形成了类的层次结构。

现实世界中的许多事物之间不是相互孤立的，它们往往具有共同的特征，也存在内在

的差别。人们可以采用层次结构描述这些实体之间的相似之处和不同之处。例如，动物学根据自然界动物的形态、身体内部构造、胚胎发育的特点、生理习性、生活的地理环境等特征，将特征相同或相似的动物归为同一类。根据身体中是否有脊椎可将动物分为两大门类：有脊椎动物和无脊椎动物。有脊椎动物又可以根据会不会飞、有没有乳腺、有没有鳍等分为哺乳类、鸟类、爬行类、两栖类和鱼类等。

图7.1展示了动物类别之间的层次结构。最高层的动物类别往往具有最一般、最普遍的特征，越下层的动物类别越具体，并且下层包含了上层的特征。它们之间的关系是基类与派生类之间的关系。

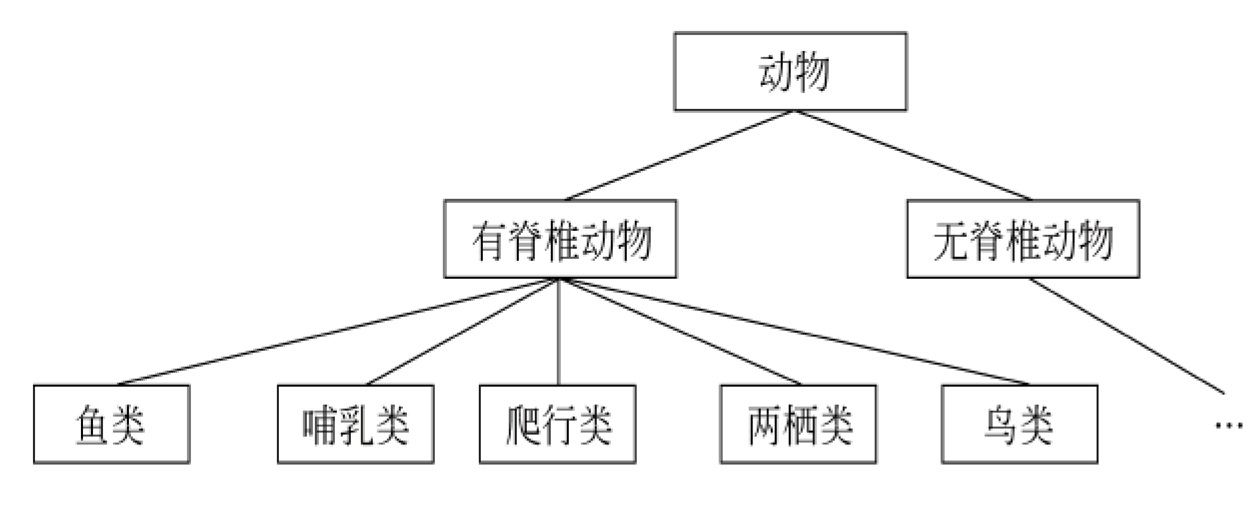


图7.1 动物的层次结构

继承可以帮助人们描述现实世界的层次关系、精确地描述事物以及理解事物的本质，是人们理解现实世界、解决实际问题的重要方法。

7.4.2 派生类的定义

派生类的定义格式如下：

```
class<派生类名>(<基类名>):
    def __init__(self[,<参数>]):
        <基类类名>.__init__(self[,<参数>])
        <新增属性定义>
```

派生类定义时必须指定基类类名，通常在类的定义时都会包含__init__()构造函数，因此在派生类中也应该先定义派生类的构造函数，由于基类的构造函数不会被自动调用，所以在派生类的构造函数中要先调用基类的构造函数，并传以必要的参数，用于初始化基类的属性，然后再通过赋值语句初始化派生类中新增加的属性成员。

【例7.12】 派生类的定义。

程序代码如下：

```
#example7.12
class Person:                                       #基类定义
    def __init__(self, name, gender,age):
        self.name=name
        self.gender=gender
        self.age=age
    def display(self):
        print('Name:',self.name,'Gender:',self.gender,'Age:',self.age)
class Student(Person):                              #派生类定义
    def __init__(self,num,major,name,gender,age):   #派生类构造函数
        Person.__init__(self,name, gender,age)      #调用基类构造函数
        self.num=num                                #派生类新增属性
        self.major=major                            #派生类新增属性
    def displayStudent(self):
        print('Number:',self.num,'Major:',self.major)
        Person.display(self)                        #调用基类方法

stu1=Student('201602181','中文','张明','男',19)
stu2=Student('201610050','软件工程','刘小天','男',20)
stu1.displayStudent()
stu2.displayStudent()
```

程序运行结果如下：

```
>>>
==============RESTART:C:/Users/python3.6/example7.12.py============
Number: 201602181 Major: 中文
Name: 张明 Gender: 男 Age: 19
Number: 201610050 Major: 软件工程
Name: 刘小天 Gender: 男 Age: 20
>>>
```

说明：

(1) 该例中 Person 为基类，Student 为派生类，派生类的构造函数__init__()负责调用基类构造函数，同时还要对派生类新增对象属性 num 和 major 进行初始化。

(2) 派生类方法 displayStudent()输出派生类新增属性值，然后再调用基类方法 display()输出基类的对象属性。

派生类在调用基类方法时，一般采用非绑定的类方法，即通过类名访问基类的方法，

并在参数列表中引入对象 self,从而达到调用基类方法的目的,如例 7.12 中的 Person.__init__(self,name, gender,age)和 Person.display(self)。但这种方式也有缺点,那就是当基类的类名改动或者派生类改为继承其他类时,在派生类中通过类名调用基类方法处的所有类名都需要修改,工作量很大。为了解决这个问题,Python 增加了内置函数 super()调用基类方法,如将例 7.12 中的 Person.__init__(self,name, gender,age)改为 super(Student,self).__init__(name, gender,age),将 Person.display(self)改为 super(Student,self).display(),也可得到相同的结果,具体参见例 7.13。

【例 7.13】 内置函数 super()的使用。

程序代码如下:

```
#example7.13
class Person:                                   #基类定义
    def __init__(self, name, gender,age):
        self.name=name
        self.gender=gender
        self.age=age
    def display(self):
        print('Name:',self.name,'Gender:',self.gender,'Age:',self.age)
class Student(Person):                                    #派生类定义
    def __init__(self,num,major,name,gender,age):         #派生类构造函数
        super(Student,self).__init__(name, gender,age)    #调用基类构造函数
        self.num=num                                      #派生类新增属性
        self.major=major                                  #派生类新增属性
    def displayStudent(self):
        print('Number:',self.num,'Major:',self.major)
        super(Student,self).display()                     #调用基类方法

stu1=Student('201602181','中文','张明','男',19)
stu2=Student('201610050','软件工程','刘小天','男',20)
stu1.displayStudent()
stu2.displayStudent()
```

程序运行结果如下:

```
>>>
==============RESTART:C:/Users/python3.6/example7.13.py============
Number: 201602181 Major: 中文
Name: 张明 Gender: 男 Age: 19
```

```
Number: 201610050 Major: 软件工程
Name: 刘小天 Gender: 男 Age: 20
>>>
```

使用 super()内置函数调用基类中的方法，当基类的名称改变或者派生类改为继承其他类时，只需修改派生类继承基类的名称即可。这样既将代码的维护量降到最低，又提高了程序开发周期。

7.4.3 派生类的组成

派生类中的属性和方法包括从基类继承的属性和方法，以及在派生类中新增加的属性和方法两部分。从基类继承的属性和方法体现了派生类从基类继承而获得的共性，而新增加的属性和方法则体现了派生类的个性。

基类成员
派生类成员

图 7.2 派生类对象结构

派生类新增加的属性和方法既体现了派生类和基类的不同，也体现了不同派生类之间的区别。派生类对象包括两个部分：一部分是基类成员，另一部分是派生类成员，如图 7.2 所示。

派生类在构造的过程中，根据实际需求主要完成以下三部分工作。

1. 从基类接收属性和方法

基类的全部成员，包括所有属性和方法都被派生类继承，作为派生类成员的一部分。

这种继承方式可能会产生数据冗余现象，因为有些基类的属性和方法虽然继承过来了，但是在派生类中却用不到。尤其是在多次派生后，会在许多派生类对象中存在大量无用的数据，不仅浪费了大量空间，而且在对象的建立、赋值、复制和参数的传递中花费了很多无谓的时间，也降低了效率。需要注意的是，正是由于这种现象的存在，在构造派生类时就需要慎重选择基类，派生类有更合理的结构，从而保证数据冗余量的最小。

2. 调整基类属性和方法

基类的属性和方法不能有选择地继承，但是却可以对这些属性和方法进行调整。最简单的方式是通过在派生类中定义新属性和方法取代基类中的属性和方法，可以在派生类中声明一个与基类同名的属性和方法，这样在派生类中的新属性和新方法将会覆盖基类的同名属性和方法。但是需要注意的是，如果重新定义方法，不仅方法名要相同，而且方法的参数表也要相同，否则方法即为重载而不是覆盖了。

3. 定义新增属性和方法

在派生类中增加新的属性和方法体现了派生类对基类功能的扩展,在定义时需仔细考虑,精心设计。

7.4.4 多继承

继承是指一个派生类从一个基类继承而来,称为单继承,但在实际工作中常常有这样的情况出现:一个派生类有两个或多个基类,派生类从两个或多个基类中继承所属的属性和方法,例如学生助教同时具有学生和教师的特征;苹果梨是苹果和梨的嫁接产物,具有两者的属性等。Python 允许一个派生类同时继承多个基类,称为多继承(Multiple Inheritance)。

【例 7.14】 多继承。

程序代码如下:

```
#example7.14
class Student():                                    #基类 Student
    def __init__(self,num,name,gender):
        self.num=num
        self.name=name
        self.gender=gender
    def displayStudent(self):
        print('学号:%s,姓名:%s,性别:%s'%(self.num,self.name,self.gender))
class Teacher():                                    #基类 Teacher
    def __init__(self,title,major,subject):
        self.title=title
        self.major=major
        self.subject=subject
    def displayTeacher(self):
        print('职称:%s,专业:%s,课程:%s'%(self.title,self.major,self.subject))

class Assistant(Student,Teacher):                   #派生类 Assistant
    def __init__(self,num,name,gender,title,major,subject,salary):
        Student.__init__(self,num,name,gender)
        Teacher.__init__(self,title,major,subject)
        self.salary=salary
    def displayAssistant(self):
```

```
        super(Assistant,self).displayStudent()
        super(Assistant,self).displayTeacher()
        print('工资',self.salary)

ta=Assistant('20150709','刘小阳','女','助教','软件工程','程序设计',800)
ta.displayAssistant()
```

程序运行结果如下：

```
>>>
==============RESTART:C:/Users/python3.6/example7.14.py============
学号:20150709,姓名:刘小阳,性别:女
职称:助教,专业:软件工程,课程:程序设计
工资 800
>>>
```

说明：

(1) 该程序首先定义 Student 和 Teacher 两个类，Student 类有 num、name 和 gender 三个属性，一个构造函数__init__()用于初始化属性，一个 displayStudent()方法用于输出学生信息；Teacher 类有 title、major 和 subject 三个属性，一个构造函数__init__()用于初始化 Teacher 类的属性，一个 displayTeacher 方法用于输出教师信息。

(2) Assistant 类是派生类，其基类有二个：Student 和 Teacher，因此属于多继承产生的派生类，该派生类新增了属性 salary，又定义了自己的构造函数和 displayAssistant()方法。

(3) 注意派生类中的构造函数不能使用 super()内置函数的方式调用基类构造函数，但是在 displayAssistant()方法中却可以使用 super()内置函数的方式调用基类的方法，这是因为派生类有二个基类，且这两个基类有同名的方法，通过 super()函数不能分辨出到底调用哪一个基类的方法，而且如果参数的个数相同，则很有可能出现某些基类的方法被多次调用，而某些基类的方法一次都没有调用的情况，需要避免出现这种情况。

7.5 多态性

多态性是面向对象程序设计的关键技术之一，与继承和类的封装构成了面向对象技术的三大特性。多态是指基类的同一个方法在不同派生类中具有不同的表现和行为，派

生类继承了基类的行为和属性后，还会增加某些特定的行为和属性，同时还可能对继承来的某些行为进行一定的改变，这都是多态的表现形式。Python 通过方法重载和运算符重载两种方式实现多态性。

7.5.1 方法重载

方法重载就是在派生类中使用与基类完全相同的方法名，从而重载基类的方法。

【例 7.15】 方法重载。

程序代码如下：

```
#example7.15
class Animal():                          #基类 Animal
    def display(self):
        print('I am an animal.')
class Dog(Animal):                       #派生类 Dog
    def display(self):                   #方法重写
        print('I am a dog.')
class Cat(Animal):                       #派生类 Cat
    def display(self):                   #方法重写
        print('I am a cat.')
class Wolf(Animal):                      #派生类 Wolf
    def display(self):                   #方法重写
        print('I am a wolf.')

x=[item() for item in (Animal,Dog,Cat,Wolf)]
for item in x:
    item.display()
```

程序运行结果如下：

```
>>>
==============RESTART:C:/Users/python3.6/example7.15.py============
I am an animal.
I am a dog.
I am a cat.
I am a wolf.
>>>
```

说明：该例中定义了基类 Animal，三个派生类 Dog、Cat 和 Wolf 均继承于基类 Animal，每个派生类中都重新定义了用于显示信息的方法 display()，而这些派生类中的方法都覆盖了基类中的方法 display()。

7.5.2 运算符重载

Python 语言提供了运算符重载功能，增强了语言的灵活性。在 Python 中除了构造函数和析构函数以外，还有大量内置的特殊方法，运算符重载就是通过重写这些内置方法实现的。这些特殊方法都是以双下画线开头和结尾的，Python 通过这种特殊的命名方式拦截操作符，以实现重载。当 Python 的内置操作运用于类对象时，会搜索并调用对象中指定的方法完成操作。

类可以重载加减运算、打印、函数调用、索引等内置运算，如表 7.2 所示。

表 7.2 Python 类特殊方法

方 法	重 载	调 用
__init__	构造函数	对象建立：X=Class(args)
__del__	析构函数	X 对象收回
__add__	运算符+	如果没有_iadd_，X+Y，X+=Y
__or__	运算符|(位 OR)	如果没有_ior_，X|Y，X|=Y
__repr__，__str__	打印、转换	print(X)、repr(X)，str(X)
__call__	函数调用	X(*args，**kargs)
__getattr__	点号运算	X. undefined
__setattr__	属性赋值语句	X. any=value
__delattr__	属性删除	delX. any
__getattribute__	属性获取	X. any
__getitem__	索引运算	X[key]，X[i:j]，无__iter__时的 for 循环和其他迭代器
__setitem__	索引赋值语句	X[key]=value，X[i:j]=sequence
__delitem__	索引和分片删除	del X[key]，del X[i:j]
__len__	长度	len(X)，如果没有__bool__，真值测试
__bool__	布尔测试	bool(X)，真测试

续表

方　法	重　载	调　用
__lt__,__gt__,	特定的比较	X<Y,X>Y
__le__,__ge__,		X<=Y,X>=Y
__eq__,__ne__		X==Y,X !=Y
__radd__	右侧加法	Other+X
__iadd__	实地(增强的)加法	X+=Y(or else __add__)
__iter__,__next__	迭代环境	I=iter(X),next(I)
__contains__	成员关系测试	item in X(任何可迭代的)
__index__	整数值	hex(X),bin(X),oct(X),O[X],O[X:]
__enter__,__exit__	环境管理器	withobj as var:
__get__,__set__	描述符属性	X.attr,X.attr=value,del X.attr
__new__	创建	在__init__之前创建对象

【例 7.16】 运算符重载。

程序代码如下：

```
#example7.16
class Number():
    def __init__(self,a,b):
        self.a=a
        self.b=b
    def __add__(self,x):                    #重载+
        return Number(self.a+x.a,self.b+x.b)
    def __sub__(self,x):                    #重载-
        return Number(self.a-x.a,self.b-x.b)

n1=Number(10,20)
n2=Number(100,200)
m=n1+n2
p=n2-n1
print(m.a,m.b)
print(p.a,p.b)
```

程序运行结果如下：

```
>>>
==============RESTART:C:/Users/python3.6/example7.16.py============
110 220
90 180
>>>
```

说明：该例中重载了 add()和 sub()两个特殊方法，当对象进行＋或－运算时，直接调用重载的方法。



习题 7

一、判断题

1. Python 中的一切内容都可以称为对象。 ( )

2. 在一个软件的设计与开发中，所有类名、函数名、变量名都应该遵循统一的风格和规范。 ( )

3. 定义类时所有实例方法的第一个参数用来表示对象本身，在类的外部通过对象名调用实例方法时不需要为该参数传值。 ( )

4. 在面向对象程序设计中，函数和方法是完全一样的，都必须为所有参数进行传值。 ( )

5. Python 中没有严格意义上的私有成员。 ( )

6. 在 Python 中定义类时，运算符重载是通过重写特殊方法实现的。例如，在类中实现了__mul__()方法即可支持该类对象的＊＊运算符。 ( )

7. 对于 Python 类中的私有成员，可以通过“对象名._类名__私有成员名”的方式访问。 ( )

8. 如果定义类时没有编写析构函数，Python 将提供一个默认的析构函数进行必要的资源清理工作。 ( )

9. 在派生类中可以通过“基类名.方法名()”的方式来用基类中的方法。 ( )

10. Python 支持多继承，如果父类中有相同的方法名，而在子类中调用时没有指定父类名，则 Python 解释器将从左向右按顺序进行搜索。 ( )

二、程序设计

1. 设计一个类 Rectangle,要求如下。

(1) 属性:长和宽。

(2) 方法:设置长和宽 setRect(self),输出长和宽 getRect(self),求周长 getPerimeter(self)和面积 getArea(self)。

2. 设计一个类 Bank,实现银行某账号的资金往来账目管理,包括创建账号、存入和取出,分别通过三个方法实现。

3. 编写程序,声明一个学生类 Student,由学生类派生出研究生类 Master,设计属性和方法,测试类的正确性。

4. 设计一个汽车类 Vehicle,包含的数据成员有车轮个数 wheels 和车重 weight。小车类 Car 是 Vehicle 的派生类,增加载人数 passenger_load;卡车类 Truck 是 Vehicle 的派生类,增加载人数 passenger_load 和载重量 payload,每个类都有相关数据的输出方法。

5. 设计一个圆类 Circle 和一个桌子类 Table,由这两个类派生出一个圆桌类 Roundtable,要求输出一个圆桌的高度、面积和颜色等数据。

第 8 章

Python 文件处理

学习目标

- 掌握文件的基本概念，了解文件的分类
- 掌握文件的打开、关闭、读/写以及定位等的使用方法
- 掌握 Python 内置的 os 模块常用函数的使用方法



8.1 文件的概念

Python 的输入/输出功能可以通过多种方式实现，既可以将终端作为对象，从终端键盘输入数据，将运行结果输出到终端上，也可以采用文件的方式进行。文件是一个用于存储数据的常用媒介，在实际应用程序的开发过程中常会涉及对文件的操作。

8.1.1 文件

文件是指存储在外部介质上的数据的集合，一批数据是以文件的形式存放在外部介质上的（如硬盘、U 盘），而操作系统正是以文件为单位对数据进行管理的。若想访问存储在文件中的数据，一般的操作是先按文件名找到所指定的文件，然后再从该文件中读取数据。若要将数据存储在外部介质中，也必须先建立一个以文件名作为标识的文件，才能向它输入数据。

从操作系统的角度来看，常用的终端输入/输出设备也是文件，终端键盘是一个输入文件，显示器和打印机是输出文件。在程序运行过程中，经常需要将一些中间数据或最终的结果输出到外部介质上存储起来，以后需要时再取出来输入到内存中。

8.1.2 文件的分类

根据文件编码方式的不同，Python 将文件分为文本文件和二进制文件。

1. 文本文件

文本文件也称为 ASCII 码文件，由 ASCII 码字符组成且不带任何格式，通常使用文本处理软件编辑。文本文件的读取必须从文件的头部开始，一次全部读出，不能只读取中间的一部分数据，不可以跳跃式访问。文本文件的每一行文本相当于一条记录，每条记录可长可短，记录之间使用换行符进行分隔，不能同时进行读操作和写操作。

文本文件的优点是使用方便，占用内存资源较少，但其访问速度较慢，不易维护。

2. 二进制文件

二进制是最原始的文件类型，直接把二进制码存放在文件中，以字节为单位访问数据，不能用文本处理软件编辑。二进制文件允许程序按所需的任何方式组织和访问数据，也允许对文件中的各字节数据进行存取和访问。

根据存储数据的性质可以将文件分为程序文件和数据文件；根据文件的流向分为输入文件和输出文件。



8.2 文件的打开与关闭

文件的访问主要是对文件进行读、写操作，但在文件读/写前必须打开文件，而在使用结束后应关闭文件。

8.2.1 文件的打开

在 Python 语言中，可以用 open()和 file()内置函数打开文件，二者具有相同的功能，可以相互替代。本书以 open()为例进行讲述。

open()函数打开文件的格式如下：

```
<文件对象名>=open(<文件名>[,<访问方式>][,<缓冲方式>])
```

其中，文件名指定了需要打开的文件；访问方式是一个可选参数，表示打开文件的方式，其值是一个字符串，默认值为'r'，即只读方式，所有的打开方式如表 8.1 所示；缓冲方式也是一个可选参数，用于按要求访问文件所采用的缓冲方式，0 表示不缓冲，1 表示只缓冲一行数据，任何其他大于 1 的值表示使用给定值作为缓冲区大小，给定负值代表使用系统默认缓冲机制，该参数的默认值为-1。

表 8.1 文件打开方式

模 式	描 述
r	打开一个文件为只读方式,文件指针置于文件的开头,为默认模式
rb	打开一个文件为二进制只读方式,文件指针置于文件的开头,为默认模式
r+	打开一个文件为读/写方式,文件指针置于文件的开头
rb+	打开一个文件为二进制读/写方式,文件指针置于文件的开头
w	打开一个文件为只写方式,如果文件存在则覆盖该文件;如果文件不存在,则创建写入新文件
wb	打开一个文件为二进制只写方式,如果文件存在则覆盖该文件;如果该文件不存在,则创建写入新文件
w+	打开一个文件为读/写方式,如果文件存在则覆盖现有文件;如果该文件不存在,则创建用于读/写操作的新文件
wb+	打开一个文件为二进制读/写方式,如果文件存在则覆盖现有文件;如果该文件不存在,则创建用于读/写操作的新文件
a	打开一个文件为追加方式,如果文件存在,则文件指针指向该文件的末尾,该文件以追加模式打开;如果该文件不存在,则创建一个用于写入的新文件
ab	打开一个文件为二进制追加方式,如果文件存在,则文件指针指向该文件的末尾,该文件以追加模式打开;如果该文件不存在,则创建一个用于写入的新文件
a+	打开一个文件为追加和读/写方式,如果文件存在,则文件指针指向该文件的末尾,该文件以追加模式打开;如果该文件不存在,则创建一个用于写入的新文件
ab+	打开一个文件为二进制追加和读/写方式,如果文件存在,则文件指针指向该文件的末尾,该文件以追加模式打开;如果该文件不存在,则创建读/写操作的新文件

说明:

(1) 用'r' 方式打开文件只能读取文件中的数据,不能向该文件写入数据,而且该文件必须已经存在,不能用'r' 方式打开一个并不存在的文件,否则会抛 IOError 异常,提示该文件不存在。

(2) 用'w' 方式打开文件只能用于向该文件写入数据,不能读取其中的数据,如果原来不存在该文件,则在打开时新建一个以指定名字命名的文件,如果原来已有一个该名字的文件,则在打开时将该文件删除,然后重新建立一个新文件。

(3) 用'a' 方式打开文件是在打开文件后允许向文件末尾添加新的数据,而不删除原有数据,使用'a' 方式打开文件要求该文件必须已经存在,否则会抛 IOError 异常。

(4) 所有带'+' 方式打开的文件既可以读又可以写,用'r+' 方式时该文件应该已存在,能够读取其中的数据;用'w+' 方式时则新建一个文件,先向该文件写入数据,然后可以读取此文件中的数据;用'a+' 方式打开的文件,原来的文件不被删除,位置指针移到文

件末尾，可以添加，也可以读取。

(5) 如果不能完成文件打开的操作，open 函数会抛出一个 IOEroor 异常，出错原因可能是用'r+' 方式打开一个并不存在的文件，或是磁盘出现故障，或是磁盘已满无法建立新文件等，因此，通常使用异常处理(try)保证文件安全。

(6) 在读取文本文件的数据时，将回车换行符转换为单个换行符，在向文件写入数据时把换行符转换成回车和换行两个字符，用二进制文件时不进行这种转换，内存中的数据与输出文件完全一致。

(7) 在程序开始运行时，系统自动打开三个标准文件：标准输入(stdin)、标准输出(stdout)和标准错误输出(stderr)，通常这三个文件都与终端相联系，因此从终端输入或输出时都不需要打开终端文件。

8.2.2　文件的关闭

当一个文件使用完后应该关闭，以防止它再次被误用。文件关闭就是使指向该文件对象的引用不再指向该文件，以后不能再通过该引用对原来与其相联系的文件进行读/写操作，除非再次打开，使该文件引用变量重新指向该文件。Python 利用 close()函数关闭文件，其格式如下：

```
<文件对象名>.close()
```

例如：

```
f=open('E:\name.txt','r+')
    …
f.close()
```

其功能就是通过 open()函数返回文件对象赋给变量 f，使 f 指向所打开的文件对象，然后对文件进行操作，最后执行 f 的 close()函数，关闭该文件，即变量 f 不再指向该文件。

【例 8.1】 文件的打开与关闭。

程序代码如下：

```
#example8.1
try:
    fh=open("testfile.txt", "w")
    fh.write("这是一个测试文件,用于测试异常!!\n")
    fh.write("这是用于测试文件操作!!")
except IOError:
```

```
    print("Error: 没有找到文件或读取文件失败")
finally:
    print("内容写入文件成功")
    fh.close()
```

程序运行结果如下：

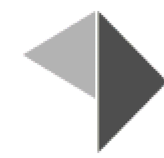
```
>>>
==============RESTART:C:/Users/python3.6/example8.1.py=============
内容写入文件成功
>>>
```

说明：

(1) 该例中以'w' 方式打开当前文件夹下的 testfile.txt 文件，如果该文件不存在，则创建以 testfile 命名的文件，然后在文件中写入字符串"这是一个测试文件，用于测试异常!!"。

(2) 以这种方式打开的文件只能进行写入操作，否则会抛出异常。

(3) 文件的打开操作及其读/写操作都放在 try 语句块中，然后在 except 子句捕获异常，最后在 finally 子句中关闭所打开的文件。



8.3 文件的读/写

8.3.1 文件的读取

文件内容的读取可以通过三个函数实现：read()、readline()和 readlines()，但文件的读取方式各不相同。

1. read()函数

read()函数可以一次性地将文件中的所有内容全部读取，也可以指定每次读多少个字节。函数调用格式如下：

```
<变量>=<文件对象>.read([size])
```

其中，变量用以存放从文件中读取的内容，size 参数表示读取文件中的前几个字符的数据，该参数是一个可选参数，不指定或指定负值(系统默认值为－1)，将读取文件的所有

内容。

【例 8.2】 read()函数的使用。

程序代码如下：

```
#example8.2
f=open("testfile.txt",'r')
fcontent=f.read()
print(fcontent)
f.seek(0)                          #位置指针移回到文件头
fcon=f.read(8)
print(fcon)
f.close()
```

程序运行结果如下：

```
>>>
==============RESTART:C:/Users/python3.6/example8.2.py=============
这是一个测试文件,用于测试异常!!
这是用于测试文件操作!!
这是一个测试文件
>>>
```

说明：

(1) 当不指定要读取的字节数，且当前文件位置指针指向该文件头时，默认读取所有的文件内容，所以输出内容和文件内容一致。

(2) 文件读取结束，位置指针移到文件内容最后一个字节的后面，需要使用 seek()函数把位置指针移到文件内容的起始处，否则无法继续读取文件内容。

(3) f.read(8)的含义即为从文件头开始读取 8 个字符的内容。

2. readline()函数

readline()函数也可以读取文件的内容，但它的读取方式不同于 read()函数，它每次只读取文件中的一行数据，调用格式如下：

```
<变量>=<文件对象>.readline([size])
```

其中，变量用以存放从文件中读取的内容，size 参数也是一个可选参数，但它表示读取文件中当前位置指针指向的行的前几个字节的数据，不指定或指定负值(系统默认值

为-1),将读取当前位置指针指向的行的所有内容。

【例 8.3】 readline()函数的使用。

程序代码如下:

```
#example8.3
f=open("testfile.txt",'r')
fcontent=f.readline()
print(fcontent)
fcon=f.readline(6)
print(fcon)
f.close()
```

程序运行结果如下:

```
>>>
==============RESTART:C:/Users/python3.6/example8.3.py=============
这是一个测试文件,用于测试异常!!

这是用于测试
>>>
```

说明:

(1) 当刚打开文件时,位置指针指向第一行开头,不指定读取当前行的前几个字符,系统默认读取当前行的所有字符,包含换行符,因此输出结果为“这是一个测试文件,用于测试异常!!”,且输出换行符。

(2) 当位置指针指向第二行开头,指定读取当前位置指针指向的行的前 6 个字符数据时,则输出“这是用于测试”。

3. readlines()函数

与 read()和 readline()函数不同,readlines()函数是一次性读取当前位置指针指向处后面的所有内容,函数返回的是一个由每行数据组成的一个列表。函数调用格式如下:

```
<变量>=<文件对象>.readlines()
```

注意,该函数没有参数,一般使用循环的方式读取文件中的内容。

【例 8.4】 readlines()函数的使用。

程序代码如下:

```
#example8.4
f=open("testfile.txt",'r')
fcontent=f.readlines()
for oneline in fcontent:
    print(oneline)
f.close()
```

程序运行结果如下：

```
>>>
==============RESTART:C:/Users/python3.6/example8.4.py=============
这是一个测试文件,用于测试异常!!

这是用于测试文件操作!!
>>>
```

说明：readlines()函数自动将文件内容分析成一个行的列表,该列表由循环处理,使用起来比 readline()函数速度快。

8.3.2　文件的写入

将数据写入文件可以通过两个函数实现：write()和 writelines(),这两个函数的区别在于操作的对象不同,write()函数是将一个字符串写入到文件中,而 writelines()函数是将列表中的字符串内容写入到文件中。

1. write()函数

write()函数的功能是将字符串写入文件,在使用该函数前,打开文件函数 open()不能以'r'的方式使用。write()函数的调用格式如下：

```
<文件对象>.write(<变量>)
```

其中,变量表示要写入的内容,可以是一个字符串或指向字符串对象的变量,也可以是字符串表达式。

【例 8.5】　write()函数的使用。

程序代码如下：

```
#example8.5
f=open("myfile.txt",'w+')
f.write("这是一个测试文件,用于测试文件的写入!")
f.write('\n')
f.write("文件写入可以使用write函数")
f.seek(0)
fc=f.read()
print(fc)
f.close()
```

程序执行后,运行结果如下:

```
>>>
=============RESTART:C:/Users/python3.6/example8.5.py============
这是一个测试文件,用于测试文件的写入!
文件写入可以使用write函数
>>>
```

说明:

(1) 程序第一次调用write()函数将“这是一个测试文件,用于测试文件的写入!”字符串写入文件myfile.txt,此时位置指针指向最后一个字符(!)的后面,第二次调用write()函数则直接写入换行符,此时位置指针位于第二行开头,第三次调用write()函数时则把“文件写入可以使用write()函数”字符串写入,此时位置指针位于最后一个字符的后面。

(2) 调用seek()函数将位置指针重新指向文件头,再通过read()函数读取文件中的内容并输出。

2. writelines()函数

writelines函数也可以对文件进行写入操作,其功能是将一个列表的内容都写入到文件中。函数调用格式如下:

```
<文件对象>.writelines(<列表>)
```

其中,参数列表为字符串列表,函数writelines()将字符串列表写入文件中。

【例8.6】 writelines()函数的使用。

程序代码如下:

```
#example8.6
f=open("myfile.txt",'w+')
strlist=["这是一个测试文件,用于测试文件的写入!\n","文件写入可以使用 write 函数"]
f.writelines(strlist)
f.seek(0)
fc=f.read()
print(fc)
f.close()
```

程序运行结果如下：

```
>>>
==============RESTART:C:/Users/python3.6/example8.6.py=============
这是一个测试文件,用于测试文件的写入!
文件写入可以使用 write 函数
>>>
```

说明：该例中定义了一个包含两个字符串元素的列表，将字符串列表通过 writelines()函数写入文件中。然后调用 seek()函数，把位置指针重新指向文件头，再通过 read()函数读取文件中的内容并输出。



8.4　文件的定位

文件中有一个位置指针，指向当前读/写位置。如果顺序读/写一个文件，每次读/写一个字符，读/写完成后该指针自动移动指向下一个字符。Python 提供了 seek()函数可以实现随机读/写文件的功能；另外，Python 还提供了 tell()函数，其功能是获取位置指针的当前位置。

8.4.1　seek()函数

对文件可以进行顺序读/写，也可以进行随机读/写。如果位置指针是按字节位置顺序移动，就是顺序读/写；如果能将位置指针按需要移动到任意位置，就可以实现随机读/写，也就是读/写文件中任意位置上的字符。

使用 seek()函数可以改变文件的位置指针，其调用格式如下：

```
<文件对象>.seek(<偏移量>,<起始点>)
```

其中,参数起始点可以选择 0、1 或 2,0 表示文件头,1 表示当前位置,2 表示文件尾;偏移量指以起始点为基点,往后移动的字符数。

【例 8.7】 seek()函数的使用。

程序代码如下:

```
#example8.7
f=open("myfile.txt",'w+')
strlist=['abc','edf','ghi']
f.writelines(strlist)
f.seek(0)
fc1=f.read(1)
print(fc1)
f.seek(0,1)
fc2=f.read(1)
print(fc2)
f.seek(5)
fc3=f.read(1)
print(fc3)
f.seek(0,2)
f.write('xyz')
f.seek(0)
fc=f.read()
print(fc)
f.close()
```

程序运行结果如下:

```
>>>
=============RESTART:C:/Users/python3.6/example8.7.py============
a
b
f
abcedfghixyz
>>>
```

说明:

(1) 该例中第一次调用函数 seek(0),表示把位置指针移到文件头,然后读取一个字

节的数据并输出，结果为显示字符'a'；第二次调用函数 seek(0,1)，表示把位置指针移到以当前位置为基准，偏移量为 0 的位置，即指针不动，指向字符'b'，然后读取输出；第三次调用函数 seek(5)，表示将位置指针移到文件头后第 5 个字符的位置，指向字符'f'，读取并输出；第四次调用 seek(0,2)，表示将位置指针指向文件尾，然后在文件尾添加新内容；第五次调用函数 seek(0)，再将位置指针移到文件头，读取文件并输出。

(2) 注意，seek()函数在使用时，若起始点为 1 或 2，偏移量只能设置为 0。

8.4.2　tell()函数

tell()函数的功能是获取文件位置指针的当前位置，函数返回值即为当前位置，用相对于文件头的位移量表示。tell()函数调用格式如下：

```
<文件对象>.tell()
```

【例 8.8】 tell()函数的使用。

程序代码如下：

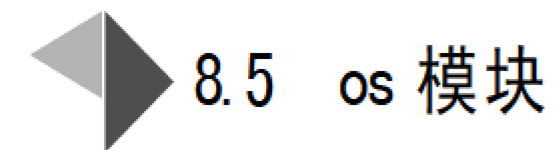
```
#example8.8
f=open("myfile.txt",'w+')
strlist=['abc','edf','ghi']
f.writelines(strlist)
print("当前位置指针：",f.tell())
f.seek(0)
print("当前位置指针：",f.tell())
fc1=f.read(1)
print(fc1)
f.seek(0,1)
print("当前位置指针：",f.tell())
fc2=f.read(1)
print(fc2)
f.seek(5)
print("当前位置指针：",f.tell())
fc3=f.read(1)
print(fc3)
f.seek(0,2)
print("当前位置指针：",f.tell())
f.write('xyz')
```

```
f.seek(0)
print("当前位置指针：",f.tell())
fc=f.read()
print(fc)
f.close()
```

程序运行结果如下：

```
>>>
==============RESTART:C:/Users/python3.6/example8.8.py=============
当前位置指针：9
当前位置指针：0
a
当前位置指针：1
b
当前位置指针：5
f
当前位置指针：9
当前位置指针：0
abcedfghixyz
>>>
```

说明：由于文件中的位置指针经常移动，不容易知道当前位置，因此用 tell()函数得到位置指针当前位置非常实用。



8.5 os 模块

Python 内置的 os 模块直接调用操作系统提供的接口函数，用于对目录和文件进行操作。当导入 os 模块时使用 import os。Python 提供了丰富的方法用来处理文件和目录，如表 8.2 所示。

表 8.2 os 模块常用方法

方　法	功　能
os.chdir(path)	改变当前工作目录
os.close(fd)	关闭文件

续表

方　法	功　能
os. curdir	返回当前目录
os. dup(fd)	复制文件
os. dup2(fd，fd2)	将一个文件复制到另一个文件
os. getcwd()	返回当前工作目录
os. listdir(path)	返回 path 指定的文件夹包含的文件或文件夹的名字的列表
os. makedirs(path[，mode])	创建多级目录的文件夹
os. mkdir(path[，mode])	创建一级目录的名为 path 的文件夹
os. open(file，flags[，mode])	打开一个文件
os. path. abspath(path)	返回 path 规范化的绝对路径
os. path. basename(path)	返回 path 最后的文件名
os. path. isabs(path)	如果 path 是绝对路径，则返回 True
os. path. exists(path)	如果 path 存在，则返回 True，否则返回 False
os. path. isfile(path)	如果 path 是一个存在的文件，则返回 True，否则返回 False
os. path. isdir(path)	如果 path 是一个存在的目录，则返回 True，否则返回 False
os. remove(path)	删除路径为 path 的文件
os. removedirs(path)	递归删除目录，若目录为空，则删除，并递归到上一级目录，若也为空，则删除
os. rename(src，dst)	重命名文件或目录
os. renames(old，new)	递归地对目录进行更名，也可以对文件进行更名
os. rmdir(path)	删除 path 指定的空目录，如果目录非空，则抛出一个 OSError 异常

【例 8.9】　文件的删除。

程序代码如下：

```
#example8.9
import os
filename='myfile.txt'
isExist=os.path.exists(filename)
```

```
if isExist:
    os.remove(filename)
    print("文件删除成功!")
else:
    print("所要删除的文件不存在!")
```

程序运行结果如下:

```
>>>
==============RESTART:C:/Users/python3.6/example8.9.py=============
文件删除成功!
>>>
```

说明:该删除首先调用 os 的子模块 path 的 exists()函数判断所删除的文件是否存在,如果存在则将其删除,否则提示所要删除的文件不存在。该例中由于前面已生成文件 myfile.txt,因此可以删除,但若再次运行程序,由于文件已被删除,则会显示“所要删除的文件不存在!”。

【例 8.10】 文件夹的创建。

程序代码如下:

```
#example8.10
import os
dirname='E:\test'
multipledirname='E:\test\python\program'
isExist=os.path.exists(dirname)
if isExist:
    print(dirname,'文件夹已存在')
else:
    os.mkdir(dirname)
    print('成功创建一级目录的文件夹')
isExist=os.path.exists(multipledirname)
if isExist:
    print(multipledirname,'文件夹已存在')
else:
    os.makedirs(multipledirname)
    print('成功创建多级目录的文件夹')
```

程序运行结果如下：

```
>>>
==============RESTART:C:/Users/python3.6/example8.10.py============
E:\test 文件夹已存在
成功创建多级目录的文件夹
>>>
```

说明：该程序先调用 exits()函数判断 E 盘下是否有 test 文件夹存在，如果有则提示该文件夹已存在，否则调用 mkdir()函数创建 test 文件夹；创建多级文件夹时，同样先判断其是否存在，不存在就调用 makedirs()函数创建。

【例 8.11】 文件夹的删除。

程序代码如下：

```
#example8.11
import os
import shutil
dirname='E:\program'
multipledirname='E:\test\python'
isExist=os.path.exists(dirname)
if isExist:
    os.rmdir(dirname)
    print('成功删除文件夹!')
else:
    print('文件夹不存在!')
isExist=os.path.exists(multipledirname)
if isExist:
    shutil.rmtree(multipledirname)
    print('成功删除非空文件夹!')
else:
    print('文件夹不存在!')
```

程序运行结果如下：

```
>>>
==============RESTART:C:/Users/python3.6/example8.11.py============
```

```
文件夹不存在!
成功删除非空文件夹!
>>>
```

说明:

(1) 在该例中,如果 E 盘中已有“E:\\program”文件夹和“E:\\test\\python\\program”文件夹,则会得到上述结果,将两个文件夹删除。

(2) 删除文件夹可以通过 rmdir()和 rmtree()两个函数完成,二者的区别在于 rmdir()函数只能删除空的文件夹,而 rmtree()函数则可以删除非空的文件夹。

(3) rmdir()函数属于 os 模块,而 rmtree()函数属于 shutil 模块。



习题 8

一、程序阅读

1. 阅读下面的程序,写出程序的功能。

```
import string
fp=open('test1.txt')
a=fp.read()
fp.close()
fp=open('test2.txt')
b=fp.read()
fp.close()
fp=open('test3.txt','w')
l=list(a+b)
l.sort()
s=''
s=s.join(l)
fp.write(s)
print(s)
fp.close()
```

2. 运行下面程序,若输入字符串 this is a file,请写出运行结果。

```
fp=open('test.txt','w')
string=input('please input a string:\n')
```

```
string=string.upper()
fp.write(string)
fp=open('test.txt','r')
print(fp.read())
fp.close()
```

二、程序设计

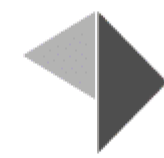
1. 从键盘输入一些字符,逐个把它们送到磁盘中,直到输入一个#为止。
2. 编写程序,将一个内容复制到另外一个文件中。

第 9 章

Python 异常处理

学习目标

- 了解程序异常产生的原因
- 掌握基本的 try…except 语句的用法
- 掌握 try…except 语句中可能的 else 子句以及 finally 子句的用法
- 理解 try 语句和 except 语句各自的功能



9.1 Python 的异常

异常(Exception)是指程序运行时引发的错误。在程序运行过程中,总会不可避免地遇到各种各样的错误。有的错误是程序编写本身造成的,如数据类型设置错误;有的错误是程序运行过程中由于用户输入错误造成的;还有一类错误是完全无法在程序运行过程中预测的,如在从网络抓取数据过程中发生网络中断等。如果不处理这些异常,则程序就会因为各种问题终止并退出。Python 内置了一整套异常处理机制,可以对异常进行处理。

9.1.1 Python 的常见异常

事实上,在前面章节的学习过程中,或者在执行命令或程序的过程中,Python 的各种异常情况曾经多次出现,只是没有详细解释这些概念。这些异常情况的出现是因为程序或命令中有错误,解释器终止了程序或命令的执行。常见的 Python 异常类型如下。

1. 除零错误(ZeroDivisionError)

这是由于错误地将数值 0 作为除数引发的异常。

```
>>>25/0
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in<module>
    25/0
ZeroDivisionError: division by zero
```

2. 变量名错误(NameError)

这是由于命令或程序中访问的变量未经定义而直接使用造成的。

```
>>>print(numbers)
Traceback(most recent call last):
  File "<pyshell#11>", line 1, in<module>
    print(numbers)
NameError: name 'numbers' is not defined
```

3. 操作数类型错误(TypeError)

这类错误主要是由于不符合表达式中运算符的运算规则或者函数参数类型的错误造成的。

```
>>>"abc"+123
Traceback(most recent call last):
  File "<pyshell#12>", line 1, in<module>
    "abc"+123
TypeError: must be str, not int
>>>len(123456)
Traceback(most recent call last):
  File "<pyshell#17>", line 1, in<module>
len(123456)
TypeError: object of type 'int' has no len()
```

4. 下标越界错误(IndexError)

请求的索引下标超出了序列的范围而造成的异常。

```
>>>list1=[1,2,3]
>>>list1[3]
Traceback(most recent call last):
  File "<pyshell#15>", line 1, in<module>
    list1[3]
IndexError: list index out of range
```

5. 打开文件错误(FileNotFoundError)

打开的文件不存在而引发的异常。

```
>>>fp=open("sample.txt","r+")
Traceback(most recent call last):
  File "<pyshell#18>", line 1, in<module>
fp=open("sample.txt","r+")
FileNotFoundError: [Errno 2] No such file or directory: 'sample.txt'
```

6. 语法错误(SyntaxError)

程序代码中的语法错误。

```
>>>sorted("[8,5,9,12"])
SyntaxError: invalid syntax
```

除此之外,还有很多其他的异常,这些错误如果得不到正确的处理,将会导致程序终止运行,甚至崩溃。异常信息给出的异常类型表明了发生异常的原因,也为程序处理异常提供了依据。

9.1.2 Python的异常处理

当程序运行过程中出现错误时,Python解释器就会指出当前程序流无法继续执行下去,会自动引发异常。异常处理就是在程序出现错误之后,为了排除错误而在正常控制流之外采取的行为。如果某段代码在程序运行期间可能会出现错误,此时建议使用异常处理结构。Python提供了多种不同形式的异常处理结构,这些处理结构都是以try…except语句的形式实现的,具体应用中可能有多种不同的形式,如:

(1) try…except

(2) try…except…else

(3) try…except…finally

(4) try…except…else…finally

具体使用哪一种结构,在实际中可根据需要选择。



9.2 常用的异常处理方法

9.2.1 基本的try…except语句

try…except语句是最基本的异常处理结构,其语法格式如下:

```
try:
    <被检测的程序代码>
except<异常类型>:
    <异常处理的程序代码>
```

try子句的代码段中包含可能出现异常的代码,except子句用来捕获异常的类型并执行异常处理。如果try子句中被检测的程序代码有异常发生且被except子句通过异常类型捕获到,则执行except子句中的异常处理程序代码;如果try子句中被检测的程序代码没有发生异常,则不执行except子句中的异常处理程序代码,程序继续向下执行;如果try子句中被检测的程序代码有异常发生,但except子句没有捕获到该异常,则程序会终止执行,并将该异常显示给最终用户。except子句中的异常类型为Python内建的异常类名称,具体内建异常体系结构如下:

```
BaseException                              所有异常类的基类
+--SystemExit                              Python解释器请求退出
+--KeyboardInterrupt                       用户中断执行(通常是输入Ctrl+C)
+--GeneratorExit                           生成器(generator)发生异常并通知退出
+--Exception                               常规错误的基类
     +--StopIteration                      迭代器没有更多的值
     +--StandardError                      所有内建标准异常的基类
     |    +--BufferError
     |    +--ArithmeticError               所有数值计算错误的基类
     |    |    +--FloatingPointError       浮点计算错误
     |    |    +--OverflowError            数值运算超出最大限制
     |    |    +--ZeroDivisionError        除(或取模)零(所有数据类型)
     |    +--AssertionError                断言语句失败
```

```
|     +--AttributeError                    对象没有这个属性
|     +--EnvironmentError                  操作系统错误的基类
|     |    +--IOError                      输入/输出失败
|     |    +--OSError                      操作系统错误
|     |         +--WindowsError (Windows)     Windows 系统调用失败
|     |         +--VMSError (VMS)
|     +--EOFError                          没有内建输入,到达 EOF 标记
|     +--ImportError                       导入模块/对象失败
|     +--LookupError                       无效数据查询的基类
|     |    +--IndexError                   序列中没有此索引(Index)
|     |    +--KeyError                     映射中没有这个键
|     +--MemoryError                       内存溢出错误
|     +--NameError                         未声明/初始化对象(没有属性)
|     |    +--UnboundLocalError            访问未初始化的本地变量
|     +--ReferenceError                    试图访问已经回收了的对象
|     +--RuntimeError                      一般运行时的错误
|     |    +--NotImplementedError          尚未实现的方法
|     +--SyntaxError                       Pythony 语法错误
|     |    +--IndentationError             缩进错误
|     |         +--TabError                Tab 和空格混用
|     +--SystemError                       一般的解释器系统错误
|     +--TypeError                         对类型无效的操作
|     +--ValueError                        传入无效的参数
|          +--UnicodeError                 Unicode 相关错误
|               +--UnicodeDecodeError      Unicode 解码时错误
|               +--UnicodeEncodeError      Unicode 编码时错误
|               +--UnicodeTranslateError   Unicode 转换时错误
+--Warning                                 警告的基类
     +--DeprecationWarning                 关于被启用的特征的警告
     +--PendingDeprecationWarning          关于构造将来语义会有改变的警告
     +--RuntimeWarning                     可疑的运行时行为的警告
+--SyntaxWarning                           可疑的语言的警告
     +--UserWarning                        用户代码生成警告
     +--FutureWarning
  +--ImportWarning                         导入模块/对象警告
  +--UnicodeWarning                        Unicode 警告
  +--BytesWarning                          Bytes 警告
+--Overflow Warning                        溢出警告
```

其中，BaseException 是所有内置异常类的基类，except 子句一般不会捕捉 BaseException 基类，而会明确指定捕捉哪种异常类。

如果没有异常处理结构，则程序代码一旦出现错误，程序将会终止，并将错误信息直接呈现给用户。

```
>>>a=[1,2,3,4,5]
>>>print(a[5])
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in<module>
    print(a[5])
IndexError: list index out of range
```

使用 try…except 语句，对可能出现错误的语句进行检测，一旦捕捉到异常，则处理该异常，此时程序不会终止执行。下面通过示例进一步理解异常处理结构的用法。

【例 9.1】 try…except 语句处理异常。

程序代码如下：

```
#example9.1 try…except 语句
a=[1,2,3,4,5]
try:
    print(a[5])
except IndexError:
    print("索引下标出界")
```

程序运行结果如下：

```
>>>
================RESTART:C:\python\example9.1.py================
索引下标出界
>>>
```

说明：

(1) try 语句检测 print(a[5])语句是否出错，如果出错，则 except 语句首先捕获错误类型，若错误类型为 IndexError，则捕获该错误，执行语句 print("索引下标出界")。

(2) 增加了 try…except 的异常处理模块后，即使程序运行出错，程序代码也不会终止，增强了代码的健壮性。

9.2.2 try…except…else 语句

try…except…else 语句可以看作为 try…except 语句的扩展，当 try 语句没有检测到异常时，执行 else 子句的代码，具体结构如下：

```
try:
    <被检测的程序代码>
except<异常类型>:
    <异常处理的程序代码>
else:
    <正常处理的程序代码>
```

【例 9.2】 将例 9.1 进行修改，并增加 else 子句，查看运行结果。

程序代码如下：

```
#example9.2 try…except…else 语句
a=[1,2,3,4,5]
try:
    print(a[4])
except IndexError:
    print("索引下标出界")
else:
    print("程序无异常")
```

程序运行结果如下：

```
>>>
=================RESTART:C:\python\example9.2.py=================
5
程序无异常
>>>
```

说明：

(1) try 语句检测 print(a[4])语句，结果没有出错，则不必由 except 语句捕获错误类型，直接执行 else 子句，执行语句 print("程序无异常")。

(2) 另外，即使 try 语句中的语句有错误，如果错误类型与 except 语句捕获的错误类

型不符，则不执行 except 子句中的代码，也不执行 else 子句中的代码，而是程序终止执行，并将该异常显示给最终用户。

【例 9.3】 根据用户输入数据的正确性，选择执行不同的语句。

程序代码如下：

```
#example9.3 try…except…else 语句
while True:
    try:
        x=int(input("请输入数据 1: "))
        y=int(input("请输入数据 2: "))
        z=x/y
    except ValueError:#传入无效的参数
        print("应全部输入数值数据!")
    else:
        print("最终结果为: ",z)
        break
```

程序运行结果如下：

```
>>>
=================RESTART:C:\python\example9.3.py=================
请输入数据 1: 6
请输入数据 2: a
应全部输入数值数据!
请输入数据 1: 6
请输入数据 2: 3
最终结果为: 2.0
>>>
```

说明：

(1) try 语句检测程序代码是否有错，如果用户输入了非数值数据，则产生 ValueError 类型错误，被 except 语句捕获，执行语句 print("应全部输入数值数据!")。

(2) 如果用户输入的都为数值数据，try 语句检测的程序代码不产生异常，直接执行 else 子句。

9.2.3　处理多重异常的 try…except 结构

有时，在同一段程序代码中，可能会出现多种异常情况，根据异常情况类型的不同，可

以采取不同的处理方式。此时,可以使用带有多个 except 子句的 try 语句结构,只要有某个 except 语句捕捉到了异常,则执行该 except 子句下的程序代码,其他的 except 子句不再进行异常的捕捉。具体结构如下:

```
try:
    <被检测的程序代码>
except<异常类型 1>:
    <异常处理的程序代码 1>
[except<异常类型 2>:
    <异常处理的程序代码 2>]
[except<异常类型 3>:
    <异常处理的程序代码 3>]
```

显而易见,该结构中也可以使用 else 子句,当 try 语句没有检测到异常时,执行 else 子句的代码。

【例 9.4】 修改例 9.3,实现对多种异常情况的处理。

程序代码如下:

```
#example9.4
while True:
    try:
        x=eval(input("请输入数据 1: "))
        y=int(input("请输入数据 2: "))
        z=x/y
    except ValueError:#传入参数错误
        print("应全部输入数值数据!")
    except ZeroDivisionError:                    #除零错误
        print("除数不能为零")
    except NameError:#变量未定义错误
        print("变量不存在")
    else:
        print("最终结果为: ",z)
        break
```

运行结果如下,数据输入的不同,产生不同类型的错误,输出不同的结果。

```
>>>
=================RESTART:C:\python\example9.4.py=================
请输入数据 1: a5
```

```
变量不存在
请输入数据 1: 6
请输入数据 2: a
应全部输入数值数据!
请输入数据 1: 9
请输入数据 2: 0
除数不能为零
请输入数据 1: 8
请输入数据 2: 4
最终结果为: 2.0
>>>
```

说明：

(1) try 语句检测程序代码是否有错，如果用户输入了非数值数据，则产生 NameError 或者 ValueError 类型错误，如果用户输入的第二个数据为数值 0，则产生 ZeroDivisionError 类型错误，except 语句捕获错误类型，根据捕获到的错误类型执行相应语句。

(2) 如果用户两次输入的都为数值数据，则 try 语句检测的程序代码不产生异常，直接执行 else 子句。

(3) 本例中，运行程序后第一次输入的数据产生 NameError 类型错误，第二次输入的数据产生 ValueError 类型错误，第三次输入的第二个数据产生 ZeroDivisionError 类型错误，第四次输入的两个数据都为数值，不产生错误。

9.2.4　try…except…finally 语句

无论 try 子句中的代码是否有异常产生，finally 子句下面的程序代码都会被执行。finally 必须是整个结构的最后一条语句，如果有 else 子句，则 else 子句必须出现在 finally 子句之前。具体结构如下：

```
try:
    <被检测的程序代码>
except<异常类型>:
    <异常处理的程序代码>
finally:
    <必定执行的程序代码>
```

【例 9.5】 try…except…finally 语句。

程序代码如下：

```
#example9.5
try:
    x=int(input("请输入数据 1: "))
    y=int(input("请输入数据 2: "))
    z=x/y
except ValueError:
    print("应全部输入数值数据!")
except ZeroDivisionError:
    print("除数不能为零")
except NameError:
    print("变量不存在")
else:
    print("最终结果为:",z)
finally:
    print("END")
```

连续三次运行程序，输入不同数据，运行结果如下：

```
>>>
=================RESTART:C:\python\example9.5.py=================
请输入数据 1: 3
请输入数据 2: a
应全部输入数值数据!
END
>>>
=================RESTART:C:\python\example9.5.py=================
请输入数据 1: 5
请输入数据 2: 0
除数不能为零
END
>>>
=================RESTART:C:\python\example9.5.py=================
请输入数据 1: 8
请输入数据 2: 2
最终结果为: 4.0
```

```
END
>>>
```

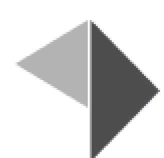
说明：

(1) 第一次运行程序产生 ValueError 类型错误，被 except 子句捕获，执行语句 print("应全部输入数值数据!")，然后直接执行 finally 中的语句。

(2) 第二次运行程序产生 ZeroDivisionError 类型错误，被 except 子句捕获，执行语句 print("除数不能为零")，然后直接执行 finally 中的语句。

(3) 第三次运行程序，try 语句没有检测到错误，直接执行 else 子句，然后再执行 finally 子句。

在 Python 的异常处理结构中，except 子句主要用来处理程序运行过程中出现的异常情况，如语法错误、向未定义的变量取值等。finally 子句主要用于无论是否发生异常情况都需要执行一些“清理工作”的场合，例如，在通信过程中，无论通信是否发生错误，都需要在通信完成或者发生错误时关闭网络连接；又如，在读一个文件时，无论是否有异常发生，最后都要关闭文件。这些场合都可以使用 finally 子句完成。



9.3 断言与上下文管理语句

断言与上下文管理语句可以实现简单的异常处理，通常结合 try…except 语句一起使用。

1. 断言语句

断言语句的格式如下：

```
assert<表达式>[,<字符串>]
```

如果<表达式>的值为 False，则抛出异常，<字符串>为异常类 AssertionError 的实例。如果<表达式>的值为 True，则什么也不做。例如：

```
>>>assert "PYTHON"=="python","'PYTHON'不等于'python'"
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in<module>
    assert "PYTHON"=="python","'PYTHON'不等于'python'"
AssertionError: 'PYTHON'不等于'python'
```

assert 语句往往与异常处理结构结合使用。

【例 9.6】 断言语句 assert。

程序代码如下：

```
#example9.6
try:
    x=int(input("请输入数据 1: "))
    y=int(input("请输入数据 2: "))
    s="两次输入的数据不相等"
    assert x==y, s
except AssertionError:
    print(s)
```

程序运行结果如下：

```
>>>
================RESTART:C:\python\example9.6.py================
请输入数据 1: 4
请输入数据 2: 6
两次输入的数据不相等
>>>
```

说明：

(1) 运行时两次输入的数据不相等，则 assert 语句中的表达式 x==y 的结果为 False，因此抛出异常。

(2) except 语句捕获异常类型为 AssertionError，执行语句 print(s)。

2. 上下文管理语句

在编程中经常遇到如下情况：有一个特殊的语句块，在执行这个语句块之前需要先执行一些准备动作；当语句块执行完成后，需要继续执行一些收尾动作，例如操作文件或数据库。上下文管理语句 with 就可以实现语句块执行前的准备动作以及执行后的收尾动作。

with 语句格式如下：

```
with<上下文管理表达式>[<as 变量>]:
    <语句块>
```

<上下文管理表达式>是支持上下文管理协议的对象，如 file、thread. LockType、threading. Lock 等，负责维护上下文环境，<as 变量>以变量方式保存上下文管理对象。

【例 9.7】 上下文管理语句。

程序代码如下：

```
#exapmle9.7
with open('hello.txt','w') as f:
    f.write('Hello')
    f.write('Python')
```

说明：

(1) 打开当前目录下的文件 hello. txt，并将这个文件对象保存为变量 f，语句块中有两条语句，实现对文件 hello. txt 的写入操作。

(2) with 上下文管理语句的功能主要体现在：文件操作完成之后可以没有 close() 语句关闭文件，当程序完成后，文件会自动关闭，避免了由于程序员忘记编写关闭文件语句而造成的程序错误。



9.4　使用 IDLE 调试代码

Python 标准开发环境 IDLE 通过调试器窗口实现代码的调试功能，解决程序出错时对错误的快速定位和调试需求。使用调试器窗口调试一个程序的步骤如下。

(1) 在 IDLE 环境下，通过 Debug 菜单下的 Debugger 命令打开调试器窗口。

(2) 打开要调试的程序并运行。

(3) 切换到调试器窗口，使用其中的控制按钮进行调试。

【例 9.8】 利用调试器调试例 6.25 的程序。

按照以上步骤打开调试器窗口后，打开程序并运行，在调试器窗口显示的状态及窗口各部分功能如图 9.1 所示。可以通过 Step 按钮实现对程序的单步执行，也可以实时查看变量的当前值，并跟踪其变化。

每种高级程序设计语言都有自己的异常处理结构，这种机制可以大幅度提高代码的健壮性。虽然异常处理结构与选择结构具有很大的相似性，很多异常处理结构也可以用选择结构代替，但是如果程序中有大量应用选择结构判断程序是否出现异常，而这些判断与程序功能的正常逻辑又不统一，这样很容易造成代码的逻辑混乱，降低代码的可读性。因此，建议在程序设计中应用异常处理结构对可能出现的程序错误做出预先的估计，做到未雨绸缪。但是，还有一点需要说明的是，尽管异常处理机制非常有效，但是也不能完全



图 9.1 用调试器调试例 6.25

依赖异常处理而忽略了对程序的常规检查。



习题 9

一、填空题

1. Python 内建异常类的基类是________。
2. Python 用来进行异常处理的语句为________。
3. 断言语句的语法为________。
4. Python 的上下文管理语句为________。

二、判断题

1. 程序中的异常处理结构在大多数情况下是没必要的。 ( )
2. 在 try…except…else 结构中,如果 try 块的语句引发了异常,则会执行 else 块中

的代码。（　　）

3. 异常处理结构中的 finally 块中的代码仍然有可能出错从而再次引发异常。（　　）

4. 带有 else 子句的异常处理结构如果不发生异常，则执行 else 子句中的代码。（　　）

5. 异常处理结构也不是万能的，处理异常的代码也有引发异常的可能。（　　）

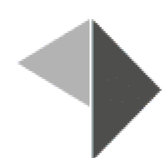
6. 在异常处理结构中，不论是否发生异常，finally 子句中的代码总会执行。（　　）

第10章

Python高级编程

学习目标

- 了解GUI编程的基本方法
- 掌握tkinter模块基本组件的用法
- 理解TCP编程和UDP编程的基本过程
- 了解网络爬虫和requests、beautifulsoup4库的基本使用方法
- 了解数据库编程的基本方法
- 掌握sqlite3库的基本使用方法



10.1 GUI编程

GUI(Graphical User Interface)的中文含义为图形用户界面或图形用户接口,是指采用图形方式显示的计算机操作用户界面。现代操作系统的图形操作界面极大地降低了计算机的使用门槛,提高了计算机的使用效率。目前,多数程序设计语言都增加了面向对象的程序设计方式,支持GUI的程序开发,与C、C++或者Java等主流程序设计语言相比,使用Python可以非常快速、简单地实现GUI编程,具有非常高的效率。

10.1.1 Python常用GUI模块

很多模块都支持Python编写GUI程序,整体上分为两大类:一类是Python自带的模块,另一类是第三方模块。tkinter是Python自带的标准GUI模块,而应用比较普遍的第三方GUI模块主要有wxPython、PyQt、Jython、IronPython等。

1. tkinter

tkinter模块是Python的标准Tk GUI工具包的接口,无须安装任何包就可以直接

使用。tkinter 可以在大多数的 UNIX 平台下使用，同样可以应用于 Windows、Linux 和 Macintosh 等多种操作系统平台。tkinter 简洁高效，适用于小型 GUI 程序的开发，Python 自带的 IDLE 就是用 tkinter 开发的。

2. wxPython

wxPython 是功能强大的支持 GUI 的 Python 模块，同时它还具有非常优秀的跨平台能力，能够支持运行在 32/64 位 Windows 和绝大多数的 UNIX 或类 UNIX 系统以及 Macintosh OS X 下。使用 wxPython 模块，Python 程序员能够轻松地创建具有健壮性、功能强大的 GUI 程序。wxPython 是 Python 语言对流行的 wxWidgets 跨平台 GUI 工具库的绑定，以 wxWidgets 的 Python 封装和 Python 模块的方式提供给用户，允许 Python 程序员很方便地创建完整的、功能键全的 GUI 用户界面。遗憾的是，wxPython 只能运行在 Python 2.x 版本上，虽然 wxPython 开发者提供了一个能够运行在 Python 3.x 版本上的最新 wxPython——wxPython Phoenix，但与 2.x 版本的 API 还是有一些差别的。

3. PyQt

PyQt 是一个 Python 编程语言和 Qt 库成功融合的 GUI 应用程序工具包。PyQt 的基础是 Qt 库，Qt 库是目前最强大的库之一，是一个跨平台的 C++ 图形用户界面库。PyQt 由 Phil Thompson 开发，实现了一个 Python 模块集，它有超过 300 个类、将近 6000 个函数和方法。PyQt 具有非常优秀的跨平台能力，可以运行在所有主流操作系统上，包括 UNIX、Windows 和 Mac。PyQt 采用双许可证，开发人员可以选择 GPL(General Public License)和商业许可。在此之前，GPL 的版本只能用在 UNIX 系统上，从 PyQt 的第 4 版开始，GPL 许可证可用于所有支持的平台。

4. Jython

Jython 是一个用于 Java 的 Python 端口，可以和 Java 无缝集成，这使 Python 脚本可以在本地机器上无缝接入 Java 类库，对于熟悉 Java 的程序员而言是一个非常好的选择。实际上，Jython 是一种完整的语言，它是一个 Python 语言在 Java 中的完全实现，Jython 不仅能提供 Python 的库，同时也提供所有的 Java 类，这使其成为一个巨大的资源库。Jython 几乎拥有标准的 Python 中不依赖于 C 语言的全部模块。例如，Jython 的用户界面将使用 Swing、AWT 或者 SWT，Jython 可以被动态或静态地编译成 Java 代码。

5. IronPython

IronPython 是 Python 编程语言和.NET平台的有机结合，能够运行在.NET和 Mono 之上，也可以运行于 Silverlight 之上。IronPython 已经很好地集成到了.NET Framework 中，Python 语言中的字符串对应于.NET的字符串对象，并且 Python 语言中对应的方法在 IronPython 中也都提供，其他数据类型也是一样。IronPython 最初由 Jim Hugunin 开发，Jim Hugunin 也是前述 Jython 模块的创造者，Jim 后来加入了微软公司，将 IronPython 作为开源软件发布。目前，微软公司仍然有一个小组在对 IronPython 进行开发。

10.1.2 tkinter 模块

tkinter 是 Python 的标准 GUI 库，可以利用 tkinter 快速创建 GUI 应用程序。由于 tkinter 是内置到 Python 的安装包，只要安装好 Python 之后就能通过 import 导入 tkinter 模块，使用起来非常方便。IDLE 本身就是用 tkinter 编写而成的，对于创建简单的图形界面，tkinter 是很好的选择。

1. 使用 tkinter 进行 GUI 编程的基本步骤

使用 tkinter 创建一个 GUI 应用程序并不复杂，主要包括以下几个步骤。

(1) 导入 tkinter 模块：

```
import tkinter 或者 from tkinter import *
```

(2) 创建 GUI 应用程序的顶层主窗口对象 top(名字任意)，用于容纳程序所有可能的组件(widget)。tkinter.Tk()返回的窗口是顶层窗口，一般命名为 root 或者 top。顶层窗口只能创建一次，并且在其他窗口创建之前被创建。例如：

```
top=tkinter.Tk()
```

(3) 在主窗口内创建其他组件，例如标签(Label)、按钮(Button)、输入框(Entry)、框架(Frame)、菜单(Menu)、滚动条(Scrollbar)等。组件既可以是独立的，也可以作为容器存在，作为容器时，它就是容器中组件的父组件。下面的代码是在主窗口中创建一个简单的标签，在标签上显示“hello，Python”：

```
label1=tkinter.Label(top,text='hello,Python')
```

(4) 将这些 GUI 模块与底层代码进行连接。将组件在窗口中显示并实现布局,最简单的布局用 pack()方法实现,grid()方法和 place()方法可以实现更复杂的布局。例如,对于上面创建的标签实例 label1,使用如下命令在主窗口显示并简单布局:

```
label1.pack()
```

(5) 进入主事件循环,响应由用户触发每个事件。组件会有一定的行为和动作,如按钮被按下、进度条被拖动、文本框被写入等,这些用户行为称为事件,针对事件的响应动作称为回调函数(callback)。用户操作,产生事件,然后相应的 callback 执行,整个过程被称为事件驱动,很显然,需要定义 callback 函数。只有窗口内的对象处于循环等待状态,才能由某个事件引发窗口内的对象完成某种功能。进入事件循环由下面的交互命令实现:

```
top.mainloop()
```

将上面五个步骤中的交互命令按照顺序合成,就可以组成一个简单的 GUI 程序。

【例 10.1】 第一个 GUI 程序。

程序代码如下:

```
#example10.1
import tkinter
top=tkinter.Tk()
label1=tkinter.Label(top,text='hello world!')
label1.pack()
top.mainloop()
```

程序运行结果如图 10.1 所示。

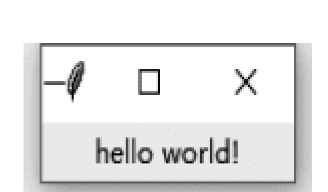


图 10.1 第一个 GUI 程序

说明:

(1) Tk 是模块 tkinter 的类,top 是 Tk 的实例,top 是最上层的组件,代表顶层窗口,其他组件都放在 top 内。

(2) label1 是类 Label 的实例,top 是标签所在的上层组件名,决定了 label1 放置在顶层窗口 top 内,text 参数用于在标签上显示文字。

(3) 调用 label1 的 pack()方法将 label1 在 top 内布局,最后利用 top 的 mainloop()方法实现主事件循环。

当然,最简单的 tkinter 程序可以只包含顶层窗口,没有任何其他组件,这样,上面的示例可以简化为如下三条语句。

【例 10.2】 只包含顶层窗口的 tkinter 程序。

程序代码如下：

```
#example10.2
import tkinter
top=tkinter.Tk()
top.mainloop()
```

该段代码会生成一个顶层窗口，没有其他组件，如图 10.2 所示。



图 10.2 最简单的 tkinter 程序

说明：创建顶层窗口 top，没有在顶层窗口内创建任何组件，然后直接进入主事件循环。

2. 公共属性

在利用 tkinter 模块进行 GUI 编程时，有很多属性是所有组件都具有的，下面进行简单描述。

(1) 尺寸

各种长度、宽度的尺寸可以用不同的单位描述。如果尺寸为整数，则默认以像素为单位，也可以指定不同的数字单位，如表 10.1 所示。

表 10.1 尺寸单位

字 符	含 义	字 符	含 义
c	厘米	m	毫米
i	英寸	p	打印机的点(即 1/27 英寸)

(2) 颜色

tkinter 中的颜色有两种不同的表示方法。

① 使用一个十六进制字符串表示颜色，该字符串指定红色、绿色和蓝色的比例。例如，#FFF 是白色的，#000000 是黑色的，#000fff000 是纯绿色，#00FFFF 是纯青色(绿加蓝)。

② 使用本地系统定义的标准颜色名称，如 white，black，red，green，blue，yellow 等。

(3) 窗口大小及位置

设置顶层窗口的大小及位置可以使用函数 geometry()，该函数的参数一般为如下形式：'wxh±x±y'。w 和 h 是以像素表达的窗口的宽度和高度，w 和 h 之间的是字符 x，作为二者之间的分隔符，注意不是叉号，也不是星号。+x 代表窗口左边距离桌面左边的

像素点距离，-x 代表窗口右边距离桌面右边的像素点距离；+y 代表窗口顶端距离桌面顶边的像素点距离，-y 代表窗口底端距离桌面底边的像素点距离。如：

```
top.geometry('400x300+150+200')
```

含义为定义一个 400 像素宽、300 像素高的窗口，窗口距离屏幕左侧 150 像素点，距离屏幕顶部 200 像素点。

3. 组件布局

组件布局(layout)就是在窗口内安排组件位置的方法。在 tkinter 中，有三种安排组件布局的方法：pack 布局、grid 布局、place 布局。

(1) pack 布局

pack 布局根据组件创建生成的顺序将组件添加到父组件中，pack 布局通过 pack()函数实现，通过设置相同的锚点(anchor)可以将组件紧挨一个位置放置，如果不指定任何选项，则默认在父窗体中自顶向下添加组件，pack()函数自动为组件分配一个合适的位置和大小。

使用 pack()函数进行布局的格式如下：

```
<组件>.pack([参数列表],…)
```

pack()函数的可选参数列表如表 10.2 所示。

表 10.2　pack()函数的可选参数列表

参数名称	描　　述	取值范围
side	指定组件停靠在父组件的哪一个方向上	top(默认值)；bottom；left；right
fill	指定水平(x)或垂直(y)方向填充 当属性 side="top"或"bottom"时，填充 x 方向 当属性 side="left"或"right"时，填充 y 方向 当 expand 选项为 yes 时，填充父组件的剩余空间	x；y；both；none
expand	当值为 yes 时，side 选项无效，组件显示在父组件中心位置；若 fill 选项为 both，则填充父组件的剩余空间	yes；no；自然数；0
anchor	指定对齐方式：左对齐"w"，右对齐 e，顶对齐 n，底对齐 s	n；s；w；e；nw；sw；se；ne；center(默认值)
ipadx，ipady	组件内部在 x(y)方向上填充的空间大小，默认单位为像素，可选单位为 c(厘米)、m(毫米)、i(英寸)、p(打印机的点，即 1/27 英寸)	非负浮点数(默认值为 0.0)

续表

参数名称	描　　述	取值范围
padx，pady	组件外部在 x(y)方向上填充的空间大小，默认单位为像素，可选单位为 c(厘米)、m(毫米)、i(英寸)、p(打印机的点，即 1/27 英寸)	非负浮点数 (默认值为 0.0)
before	将本组件于所选组建对象之前 pack，类似于先创建本组件再创建选定组件	已经 pack 后的组件对象
after	将本组件于所选组建对象之后 pack，类似于先创建选定组件再创建本组件	已经 pack 后的组件对象
in	将本组件作为所选组建对象的子组件，类似于指定本组件的 master 为选定组件	已经 pack 后的组件对象

【例 10.3】 新建程序，验证 tkinter 的 pack 布局。

程序代码如下：

```
#example10.3 pack布局示例
import tkinter
top=tkinter.Tk()
top.geometry('320x120+0+0')                    #指定主窗口的大小
label1=tkinter.Label(top, text="北京")
label2=tkinter.Label(top, text="上海")
label3=tkinter.Label(top, text="广州")
label4=tkinter.Label(top, text="深圳")
label1.pack(side='left',fill='both')
label2.pack(side='right',fill='both',padx=5,pady=3)
label3.pack(side='top',fill='x',expand='yes',anchor='n')
label4.pack(side='bottom',expand='yes',anchor='s')
top.mainloop()
```

程序运行结果如图 10.3 所示。



图 10.3　例 10.3 的运行结果

(2) grid 布局

grid 布局采用类似表格的结构组织组件，grid 布局通过 grid()函数实现。grid 采用行列确定位置，行列交汇处为一个单元格。可以将若干个单元格连接为一个更大的空间，这一操作被称为跨越，创建的单元格必须相临。每一列中，列宽由这一列中最宽的单元格确定；每一行中，行高由这一行中最高的单元格决定。组件并不是充满整个单元格，可以指定单元格中剩余空间的使用。可以空出这些空间，也可以在水平或竖直或两个方向上填满这些空间。用 grid 设计对话框和带有滚动条的窗体效果最好。

使用 grid()函数进行布局的格式如下：

```
<组件>.grid([参数列表],…)
```

grid()函数的可选参数列表如表 10.3 所示。

表 10.3 grid()函数的可选参数列表

名　称	描　　述	取值范围
row	组件所置单元格的行号	自然数(起始默认值为 0)
column	组件所置单元格的列号	自然数(起始默认值为 0)
rowspan	从组件所置单元格算起在行方向上的跨度	自然数(起始默认值为 0)
columnspan	从组件所置单元格算起在列方向上的跨度	自然数(起始默认值为 0)
ipadx,ipady	组件内部在 x(y)方向上填充的空间大小，默认单位为像素，可选单位为 c(厘米)、m(毫米)、i(英寸)、p(打印机的点，即 1/27 英寸)	非负浮点数 (默认值为 0.0)
padx,pady	组件外部在 x(y)方向上填充的空间大小，默认单位为像素，可选单位为 c(厘米)、m(毫米)、i(英寸)、p(打印机的点，即 1/27 英寸)	非负浮点数 (默认值为 0.0)
in_	将本组件作为所选组建对象的子组件，类似于指定本组件的 master 为选定组件	已经 pack 后的组件对象
sticky	组件紧靠所在单元格的某一边角	N;s;w;e;nw;sw;se;ne;center 默认为 center

【例 10.4】 新建程序，验证 tkinter 的 grid 布局。

程序代码如下：

```
#example10.4 pack 布局示例
import tkinter
top=tkinter.Tk()
```

```
top.geometry('420x180+0+0')          #指定主窗口的大小
label1=tkinter.Label(top, text="北京")
label2=tkinter.Label(top, text="上海")
label3=tkinter.Label(top, text="广州")
label4=tkinter.Label(top, text="深圳")
label5=tkinter.Label(top, text='成都')
label6=tkinter.Label(top, text='重庆')
label7=tkinter.Label(top, text='武汉')
label8=tkinter.Label(top, text='南京')
label1.grid(row=0,column=0,padx=50,pady=10)
label2.grid(row=0,column=1)
label3.grid(row=1,column=1,padx=50,pady=10)
label4.grid(row=1,column=2)
label5.grid(row=2,column=2)
label6.grid(row=2,column=3)
label7.grid(row=2,column=4)
label8.grid(row=2,column=5)
top.mainloop()
```

程序运行结果如图 10.4 所示：

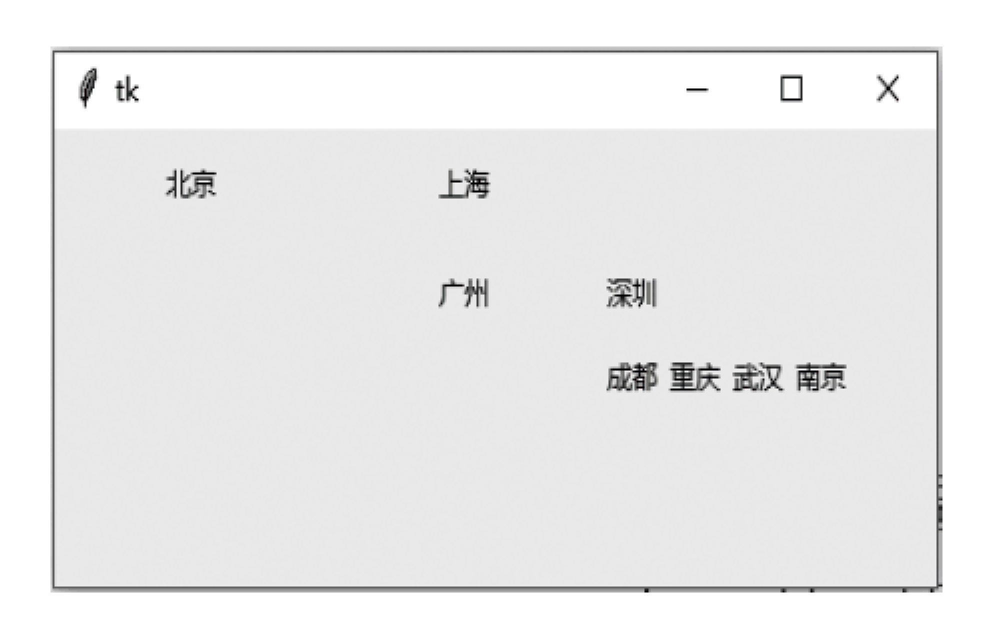


图 10.4　例 10.4 的运行结果

(3) place 布局

place()直接使用位置坐标布局，可用于更精细、更复杂的位置控制，但一般很少使用，本书不做介绍，有兴趣的读者可以参考相关资料。

4. tkinter 常用组件

tkinter 提供各种组件在一个 GUI 应用程序中使用，表 10.4 列出了常用的组件以及组件的功能描述。

表 10.4　tkinter 的常用组件

组　件	功 能 描 述
Button	按钮组件：在程序中显示按钮
Canvas	画布组件：包含图像或位图，可以显示线条或文本等图形元素
Checkbutton	多选框组件：用于在程序中提供多项选择框
Entry	输入框组件：用于接收键盘输入的数据或显示简单的文本内容
Frame	框架组件：容器类组件，在屏幕上显示一个矩形区域放置其他组件
Label	标签组件：可以显示文本或图形
Listbox	列表框组件：用来显示一个字符串列表给用户，用户可以从中做出选择
Menu	菜单组件：按下菜单按钮时弹出的菜单列表，包含多个列表项
Menubutton	菜单按钮组件：包含菜单项的组件，有下拉菜单和弹出菜单
Message	消息组件：用来显示多行文本，与 label 类似
Radiobutton	单选按钮组件：一组按钮，只有一个单选按钮可以被选中
Scale	进度条(线性滑块)组件：可以设置起始值和结束值，为输出限定范围的数字区间
Scrollbar	滚动条组件，对支持的组件提供滚动条功能，如列表框
Text	文本组件：用于显示或接收用户输入的多行文本
Toplevel	容器组件：独立的顶级窗口容器，用来提供一个单独的对话框，和 Frame 比较类似
Spinbox	输入组件：与 Entry 类似，但是可以指定输入范围值
PanedWindow	PanedWindow 是一个窗口布局管理的插件，可以包含一个或者多个子组件
LabelFrame	labelframe 是一个简单的容器组件，常用于复杂的窗口布局
tkMessageBox	用于显示应用程序的消息框

(1) 标签(Label)组件

标签组件是最简单的组件，前面的介绍中已经使用了标签组件。标签组件的格式如下：

```
<标签组件名>=tkinter.Label(<父组件>,[参数列表],…)
```

标签组件名是类 Label 的实例，如 label1，父组件是上层组件名，将在上层组件内创建一个标签组件 label1，并在适当位置显示文本或图像信息，显示信息的内容和方式受到参数列表的影响。

例如：

```
label1= tkinter.Label(top,text= 'Hello,Python!')
```

将在上层组件 top(顶层窗口)内创建一个表单组件 label1。

标签组件的参数选项如表 10.5 所示。

表 10.5 标签组件参数说明

参　数	说　明
text	标签内要显示的文字,多行以'\n'分隔
width	标签的宽度,显示文本,以单个字符大小为单位;显示图像,以像素为单位
height	标签的高度,显示文本,以单个字符大小为单位;显示图像,以像素为单位
anchor	文本或图像在背景内容区的位置：n(north),s(south),w(west),e(east),还有 ne,nw,sw,se,center(默认值),表示上北下南左西右东
background(bg)	指定背景颜色,默认值跟随系统
relief	边框样式：flat(默认)/sunken/raised/groove/ridge
borderwidth(bd)	边框的宽度,单位是像素
font	指定字体和字体大小,font=(font_name,size)
justify	指定文本对齐方式：center(默认)/left/right
foreground(fg)	指定文本(或图像)颜色
underline	单个字符添加下画线
bitmap	指定标签上显示的位图
image	指定标签上显示的图像
compound	文本和图像的位置关系。None(默认值)：显示图像不显示文本;bottom/top/left/right：图片显示在文本的下/上/左/右;center：文本显示在图片中心上方
activebacakground	设置 Label 处于活动(active)状态下的背景颜色,默认由系统指定
activeforground	设置 Label 处于活动(active)状态下的前景颜色,默认由系统指定
diableforground	指定当 Label 不可用的状态(Disable)下的前景颜色,默认由系统指定
cursor	指定鼠标经过 Label 时,鼠标的样式,默认由系统指定
state	指定 Label 的状态,用于控制 Label 如何显示。可选值有：normal(默认)/active/disable

【例 10.5】 文字标签。

程序代码如下：

```
#example10.5 文字标签示例
import tkinter
root=tkinter.Tk()
root.geometry('300x100+0+0')
root.wm_title('标签组件示例')
label1=tkinter.Label(root,text='高级语言程序设计\n--Python',\
                     height=4,width=20,relief='ridge',\
                     background='#ffffff',foreground='#ff0000',\
                     anchor='center',font='黑体',cursor='man')
label1.grid(row=0,column=1,padx=60)
root.mianloop()
```

程序运行结果如图 10.5 所示。



图 10.5　标签组件示例的运行结果

【例 10.6】 图像标签。

程序代码如下：

```
#example10.6 图像标签示例
import tkinter
root=tkinter.Tk()
root.geometry('300x100+0+0')
root.wm_title('标签组件示例')
x=tkinter.PhotoImage(file='python.gif')
label1=tkinter.Label(root,image=x,height=122,width=372,relief='ridge')
label1.pack()
root.mianloop()
```

程序运行结果如图 10.6 所示。

说明：图片文件 python.gif 必须与程序文件保存在同一目录下。



图 10.6　图像标签示例的运行结果

(2) 按钮(Button)组件

按钮可以包含文本或图像,当按钮被按下时,会调用函数或方法,引发按钮响应事件,完成相应功能。创建按钮组件的格式如下:

```
<按钮组件名>=tkinter.Button(<父组件>, [参数列表],…)
```

按钮组件名是类 Button 的实例,如 btn1,父组件是上层组件名,将在上层组件内创建一个按钮组件 btn1。举例如下:

```
btn1=tkinter.Button(top,text='Click me')
```

在上层组件 top(顶层窗口)上创建一个按钮组件 btn1。

按钮组件的参数列表选项如表 10.6 所示。其中,anchor、relief、bitmap、image、height、width、font、bg、fg、bd、cursor、underline 等参数的功能与标签组件相同,不做重复说明。

表 10.6　按钮组件参数说明

参　　数	说　　明
text	按钮上要显示的文字内容
default	默认值为 normal,如果为 disabled,则按钮不响应单击事件
takefocus	设置焦点,takefocus=1|0
state	设置组件状态;正常(normal),激活(active),禁用(disabled)
textvariable	设置文本变量
command	指定按钮的事件处理函数

按钮主要有两个方法,flash()方法和 invoke()方法。flash()方法使按钮在正常状态和激活状态之间闪烁,invoke()方法调用按钮的回调函数。

【例 10.7】 按钮组件应用示例。

程序代码如下:

```
#example10.7 按钮组件
import sys
import tkinter
top=tkinter.Tk()
top.geometry('310x180+100+100')
top.wm_title('按钮组件示例')

def prt_label():
    label1=tkinter.Label(top,text='Hello Python!')
    label1.grid(row=0,column=1)

btn1=tkinter.Button(top,text='Click me',height=2,width=12,\
                         borderwidth=5,command=prt_label)
btn2=tkinter.Button(top,text='Exit',height=2,width=12,\
                         borderwidth=5,command=sys.exit)
btn1.grid(row=1,column=0,padx=5,pady=100)
btn2.grid(row=1,column=2,padx=10)
top.mainloop()
```

程序运行结果如图 10.7 所示。



图 10.7　按钮组件示例的运行结果

说明：在创建按钮 btn1 的代码中，command=prt_label 的含义是：当按钮 bt1 被按下时，调用函数 prt_label，从而实现创建标签的动作。这种通过一个事件施加在一个对象上，而对象根据被激发的事件执行相对应程序的机制被称为事件驱动。

单击鼠标左键、中键、右键、双击鼠标、键盘上的某个键被按下都可以看作 tkinter 的事件。tkinter 中的常见事件类型如表 10.7 所示。

表 10.7 tkinter 中的事件类型

事件名称		说明
键盘事件	KeyPress	按下键盘某键时触发
	KeyRelease	按下键盘某键时触发
鼠标事件	ButtonPress	按下鼠标某键时触发
	ButtonRelease	释放鼠标某键时触发
	Motion	选中组件的同时拖曳组件移动时触发
	Enter	当鼠标指针移进某组件时，该组件触发
	Leave	当鼠标指针移出某组件时，该组件触发
	MouseWheel	当鼠标滚轮滚动时触发
窗体事件	Visibility	当组件变为可视状态时触发
	Unmap	当组件由显示状态变为隐藏状态时触发
	Map	当组件由隐藏状态变为显示状态时触发
	Expose	当组件从原本被其他组件遮盖的状态中暴露出来时触发
	FocusIn	组件获得焦点时触发
	FocusOut	组件失去焦点时触发
	Destroy	当组件被销毁时触发

例 10.7 中给出了按钮 btn1 被按下时调用的函数，按钮的其他事件被触发时可以通过将事件绑定到对象上实现，并利用回调函数调用相关函数执行相应动作。对象绑定事件的格式如下：

```
<组件对象名>.bind(<事件类型>,<回调函数>)
```

函数 bind()将<事件类型描述>的具体事件绑定到<组件对象>上，当<事件类型>描述的事件发生时，自动调用<回调函数>。事件类型以字符串形式传递且必须放置于尖括号<>内，回调函数必须定义一个形参。利用 bind()函数修改例 10.7，结果如下：

【例 10.8】 利用 bind()函数修改例 10.7。

程序代码如下：

```
#example10.8 按钮组件 bind()函数绑定
import sys
```

```
import tkinter
top=tkinter.Tk()
top.geometry('310x180+100+100')
top.wm_title('按钮组件 bind()函数绑定')

def prt_label(x):
    label1=tkinter.Label(top,text='Hello Python!')
    label1.grid(row=0,column=1)

btn1=tkinter.Button(top,text='Click me',height=2,width=12,\
                        borderwidth=5)
btn1.bind('<Button-1>',prt_label)
btn2=tkinter.Button(top,text='Exit',height=2,width=12,\
                        borderwidth=5,command=sys.exit)
btn1.grid(row=1,column=0,padx=5,pady=100)
btn2.grid(row=1,column=2,padx=10)
top.mainloop()
```

程序运行结果与例 10.7 相同,如图 10.8 所示。

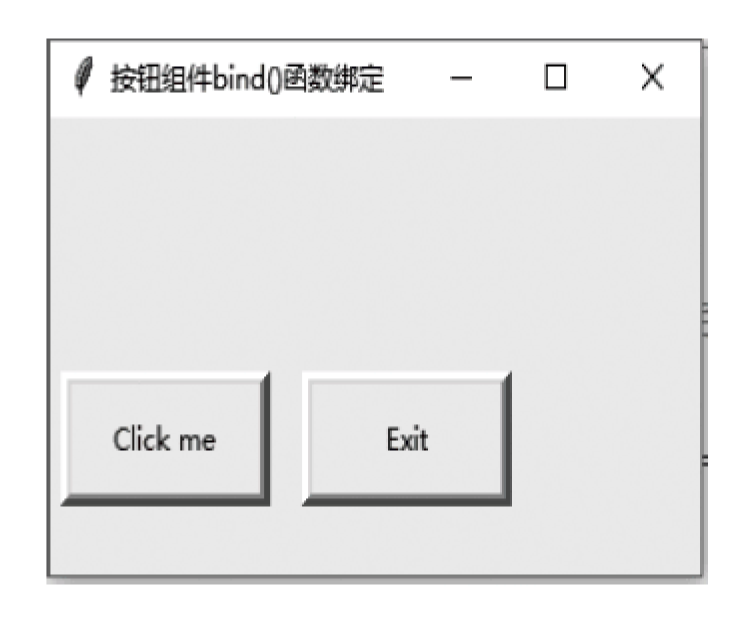


图 10.8　按钮组件 bind()函数绑定的运行结果

说明:语句 btn1.bind('<Button-1>',prt_label)将鼠标左键单击事件与按钮对象 btn1 绑定,调用回调函数 prt_label(),该回调函数包含一个形参 x,<Button-1>代表单击鼠标左键,单击鼠标中键、右键和双击鼠标左键的事件类型描述分别为<Button-2>、<Button-3>、<Double-Button-1>。

键盘某个键被按下可以类似地描述为<KeyPress-A>(A 键被按下)、<KeyPress-Y>(Y 键被按下)。还可以描述某些组合键被按下,如<Control-A>(Ctrl 键和 A 键同时被按下)、<Control-C>(Ctrl 键和 C 键同时被按下)、<Control-Shift-KeyPress-A>(Ctrl 键、Shift 键和 A 键同时被按下)。

(3) 输入框(Entry)组件和文本框(Text)组件

输入框和文本框组件都用来接收用户输入的数据并显示数据，二者的区别在于输入框接收单行数据，而文本框可以接收多行数据。输入框和文本框共享大多数的属性和方法，在此只以输入框为例介绍二者的应用。

创建输入框组件的格式如下：

```
<输入框组件名>=tkinter.Entry(<父组件>, [参数列表],…)
```

输入框组件名是类 Entry 的实例，父组件是上层组件名，将在上层组件内创建一个输入框组件，参数列表选项如表 10.8 所示。

表 10.8　输入框组件参数说明

参　　数	说　　明
insertwidth	输入框光标的宽度
insertontime	输入框光标闪烁时，显示持续时间，单位：毫秒(ms)，默认值为 600ms
insertofftime	输入框光标闪烁时，消失持续时间，单位：毫秒(ms)，默认值为 300ms
justify	当输入的文本不适应输入框时的显示方式：left/center/right，默认值为 left
show	指定输入框内容显示为字符，如显示密码可以将值设为 *
textvariable	输入框的值，是一个 StringVar()对象
xscrollcommand	建立与滚动条组件的联系，设置为滚动条组件的.set 方法

输入框的方法较多，表 10.9 只列出常用方法。

表 10.9　输入框的常用方法

方　　法	说　　明
insert(index, text)	向输入框中插入字符，index：插入位置，text：插入字符
delete(first, last)	删除输入框中从 first 开始到 last(不包含 last)的字符串，省略 last，只删除 first 位置
get()	获取输入框的字符串值
select_clear()	清除输入框选择的内容
icursor(index)	将光标移动到 index 索引位置前，文框获取焦点后成立
index(index)	返回指定的索引值，保证 index 位置上的字符是输入框最左侧的可视字符
select_range(start, end)	选中 start 索引与 end 索引之前的值，start 必须小于 end

【例 10.9】 输入框组件示例。

程序代码如下：

```
#example10.9输入框示例
import tkinter
top=tkinter.Tk()
top.geometry('300x100+50+50')
top.wm_title('输入框示例')
v=tkinter.StringVar()
en1=tkinter.Entry(top,textvariable=v)
en1.pack()
v.set('输入框示例')
top.mainloop()
```

程序运行结果如图 10.9 所示。



图 10.9 输入框示例的运行结果

说明：语句 v=tkinter.StringVar()是调用 tkinter 的 StringVar()对象，该对象用来监视在 tkinter 组件内输入的字符串类型数据，textvariable＝v 实现了将输入框内输入的字符串与 StringVar()对象的关联。

语句 v.set('输入框示例')通过调用 StringVar()对象的 set()函数在输入框内显示文本，StringVar()还有类似的函数 get()，用于返回 StringVar 变量的值。

【例 10.10】 标签、按钮、输入框组件综合应用实例。

程序代码如下：

```
#example10.10 标签、按钮、输入框综合实例
import tkinter
import sys
import re
top=tkinter.Tk()
top.geometry('400x170+350+150')
top.wm_title('标签、按钮、输入框综合实例')

def validateText():                          #有效性检验函数
    val=entry1.get()
```

```
        if re.findall('^[0-9a-zA-Z_]{1,}$',str(val)):
            return True
        else:
            label3['text']='用户名只能包含字母、数字、下画线'
            return False

def anw_button():                          #按钮响应函数
    if str.upper(entry1.get())=='PYTHON' and entry2.get()=='python_123':
        label3['text']='登录成功'
    else:
        label3['text']='用户名或密码错误,请重新输入!'

label1=tkinter.Label(top,text='用户名:',font=('宋体','18'))
label1.grid(row=0,column=0)
label2=tkinter.Label(top,text='密码:',font=('宋体','18'))
label2.grid(row=1,column=0)
v=tkinter.StringVar()                      #定义变量
entry1=tkinter.Entry(top,font=('宋体','18'),textvariable=v,\
                    validate='focusout', validatecommand=validateText)
              #建立与变量 v 的联系、有效性检验方式以及回调函数名
entry1.grid(row=0,column=1)
entry1.focus_force()                          #强行得到焦点
entry2=tkinter.Entry(top,font=('宋体','18'),show='*')
                                              #屏蔽密码
entry2.grid(row=1,column=1)
button1=tkinter.Button(top,text='登录',font=('宋体','18'),\
                        command=anw_button)
button1.grid(row=2,column=0,padx=50,pady=10)
button2=tkinter.Button(top,text='退出',font=('宋体','18'),\
                        command=sys.exit)
button2.grid(row=2,column=1,padx=80,pady=10)
label3=tkinter.Label(top,text='信息提示区',font=('华文新魏','16'),\
                    relief='ridge',width=30)
label3.grid(row=3,column=0,padx=10,pady=10,columnspan=2,sticky='s')
top.mainloop()
```

程序运行结果如图 10.10 所示。



图 10.10　标签、按钮、输入框组件综合应用实例

说明：本例中，使用 Entry 组件验证输入内容的合法性，用户名要求只能输入字母、数字和下画线的组合，输入其他字符非法。实现该功能，需要设置 validate、validatecommand 和 invalidcommand 选项。

首先启用验证的“开关”是 validate 选项，该选项可以设置的值如下。

Focus：当 Entry 组件获得或失去焦点时验证。

Focusin：当 Entry 组件获得焦点时验证。

Focusout：当 Entry 组件失去焦点时验证。

Key：当输入框被编辑时验证。

All：当出现以上任何一种情况时验证。

validatecommand 选项指定一个验证函数，该函数只能返回 True 或 False，表示验证的结果，通过 Entry 组件的 get()方法获得输入框的内容。

(4) 复选按钮(Checkbutton)组件与单选按钮(Radiobutton)组件

复选按钮也称为选择按钮,顾名思义,复选按钮用于提供多个选项供用户选择。单选按钮提供一组彼此相互排斥的选项,任何时刻只能选择一个选项,因此,单选按钮具有组的概念。创建复选按钮和单选按钮的格式分别如下:

```
<复选按钮组件名>=tkinter.Checkbutton(<父组件>, [参数列表],…)
<单选按钮组件名>=tkinter.Radiobutton(<父组件>, [参数列表], …)
```

复选按钮组件名是类 Checkbutton 的实例,单选按钮组件名是类 Radiobutton 的实例,父组件是上层组件名。除了通用属性参数外,复选按钮和单选按钮的参数列表选项如表 10.10 所示。

表 10.10 复选按钮组件和单选按钮组件的参数说明

参　数	说　明
command	当按钮状态改变时,指定按钮的事件处理函数
text	按钮标签上的文本内容
indicatoron	复选按钮的状态:设置/未设置:0;无效:1
justify	调整按钮的 text 多行文本布局:center/left/right
offvalue	当复选按钮被清除时,组件控制变量被置为 0
onvalue	当复选按钮被设置时,组件控制变量被置为 1
selectcolor	组件被设置时的颜色,默认值为 white
selectimage	组件被设置时的图片颜色
variable	复选按钮:跟踪组件的状态变量,清除为—0,设置为—1 单选按钮:控制变量,用于同组的其他单选按钮
textvariable	文本变量,用于改变按钮的标签文本内容
value	用户设置单选按钮时,控制变量的值

【例 10.11】 复选按钮与单选按钮实例。

程序代码如下:

```
#-*-coding:utf-8-*-
#example10.11
```

```
#复选按钮与单选按钮实例
import tkinter
import tkinter.font as tkFont
top=tkinter.Tk()
top.geometry('400x200+350+150')
top.wm_title('复选按钮与单选按钮实例')
def func_cb():
    if x.get()==1 and y.get()==0 and z.get()==0:
        entry1['font']=tkFont.Font(family='宋体',size=20,weight='bold')
    elif x.get()==1 and y.get()==0 and z.get()==1:
        entry1['font']=tkFont.Font(family='宋体',size=20,weight='bold',\
                                   underline=1)
    elif x.get()==1 and y.get()==1 and z.get()==0:
        entry1['font']=tkFont.Font(family='宋体',size=20,weight='bold',\
                                   slant='italic')
    elif x.get()==1 and y.get()==1 and z.get()==1:
        entry1['font']=tkFont.Font(family='宋体',size=20,weight='bold',\
                                   slant='italic',underline=1)
    elif x.get()==0 and y.get()==0 and z.get()==0:
        entry1['font']=tkFont.Font(family='宋体',size=20)
    elif x.get()==0 and y.get()==0 and z.get()==1:
        entry1['font']=tkFont.Font(family='宋体',size=20,underline=1)
    elif x.get()==0 and y.get()==1 and z.get()==0:
        entry1['font']=tkFont.Font(family='宋体',size=20,slant='italic')
    elif x.get()==0 and y.get()==1 and z.get()==1:
        entry1['font']=tkFont.Font(family='宋体',size=20,slant='italic',\
                                   underline=1)

def func_rb1():
    entry2['font']=('华文新魏',20)
def func_rb2():
    entry2['font']=('华文新魏',24)
def func_rb3():
    entry2['font']=('华文新魏',28)

v=tkinter.StringVar()                  #定义变量
entry1=tkinter.Entry(top,font=('宋体','24'),textvariable=v)
```

```
entry1.grid(row=0,column=0,columnspan=4)
entry1.focus_force()                    #强行得到焦点
w=tkinter.StringVar()
entry2=tkinter.Entry(top,font=('宋体','18'),textvariable=w)
entry2.grid(row=2,column=0,columnspan=4)
global x,y,z                            #全局变量,获取复选按钮状态值
x=tkinter.IntVar()
cb1=tkinter.Checkbutton(top,text='粗体',command=func_cb,variable=x)
cb1.grid(row=1,column=0)
y=tkinter.IntVar()
cb2=tkinter.Checkbutton(top,text='斜体',command=func_cb,variable=y)
cb2.grid(row=1,column=1)
z=tkinter.IntVar()
cb3=tkinter.Checkbutton(top,text='下画线',command=func_cb,variable=z)
cb3.grid(row=1,column=2)
rb1=tkinter.Radiobutton(top,text='20 号',command=func_rb1)
rb1.grid(row=3,column=0)
rb2=tkinter.Radiobutton(top,text='24 号',command=func_rb2)
rb2.grid(row=3,column=1)
rb3=tkinter.Radiobutton(top,text='28 号',command=func_rb3)
rb3.grid(row=3,column=2)
a=tkinter.IntVar()
rb1['variable'],rb2['variable'],rb3['variable']=a,a,a
rb1['value'],rb2['value'],rb3['value']=1,2,3
top.mainloop()
```

程序运行结果如图 10.11 所示。



图 10.11　复选按钮与单选按钮实例

(5) 框架(Frame)组件与标签框架(LabelFrame)组件

框架组件可以对其他组件进行组织和分组,它是一个容器组件,可以包含标签、输入框、复选按钮等其他组件。在屏幕上以一块矩形区域作为其他组件的容器(container)布局窗体。标签框架可以在框架周围的框架线上显示标签。创建框架和标签框架的格式如下:

```
<框架组件名>=tkinter.Frame(<父组件>, [参数列表], …)
<标签框架组件名>=tkinter.LabelFrame(<父组件>, [参数列表], …)
```

可以在框架或标签框架内创建其他组件,与直接在顶层窗口创建组件的区别就是父组件参数的不同,例如,在顶层窗口创建框架 fm,然后在框架 fm 组件内创建标签组件 label1,代码如下:

```
>>>import tkinter
>>>top=tkinter.Tk()
>>>fm=tkinter.LabelFrame(top,width=200, height=500, text='LabelFrame')
>>>fm.pack()
>>>label1=tkinter.Label(fm, text='I like Python!',bg='white')
>>>label1.pack()
>>>top.mainloop()
```

框架和标签框架的绝大多数属性与前述组件相同,在此不做重复介绍。标签控件独有的特殊属性主要有两个,分别是 labelanchor 和 labelwidget。labelanchor 属性用来控制标签在框架上的显示位置,labelwidget 指明代替标签框架周围显示的标签。

下面的示例演示在顶层窗口创建两个标签框架,并在标签框架内分别添加了单选按钮和复选框组件。

【例 10.12】 在标签框架内添加其他组件。

程序代码如下:

```
#example10.12 框架与标签框架
import tkinter
top=tkinter.Tk()
top.geometry('300x300+0+0')                    #指定主窗口大小
fm1=tkinter.LabelFrame(top,width=200, height=110,bg='white',\
                       relief='ridge',bd=5,text='字体')
fm1.grid(row=0,column=0,padx=50,pady=10)
fm1.grid_propagate(0)
fm2=tkinter.LabelFrame(top,width=200, height=130,bg='white',\
```

```
                        relief='ridge',bd=5,text='字形')
fm2.grid(row=1,column=0,padx=50,pady=10)
fm2.grid_propagate(0)                        #强迫标签框架保持原定义尺寸
x=tkinter.IntVar()
rb1=tkinter.Radiobutton(fm1,text='宋体')
rb1.grid(row=0,column=0)
rb2=tkinter.Radiobutton(fm1,text='黑体')
rb2.grid(row=1,column=0)
rb3=tkinter.Radiobutton(fm1,text='楷体')
rb3.grid(row=2,column=0)
rb1['variable'],rb2['variable'],rb3['variable']=x,x,x
rb1['value'],rb2['value'],rb3['value']=0,1,2
check1=tkinter.Checkbutton(fm2,text='加粗')
check1.grid(row=0,column=0)
check1.select()                              #该选项处于选中状态
check2=tkinter.Checkbutton(fm2,text='倾斜')
check2.grid(row=1,column=0)
check3=tkinter.Checkbutton(fm2,text='下画线')
check3.grid(row=2,column=0)
check3.select()
top.mainloop()
```

程序运行结果如图 10.12 所示。

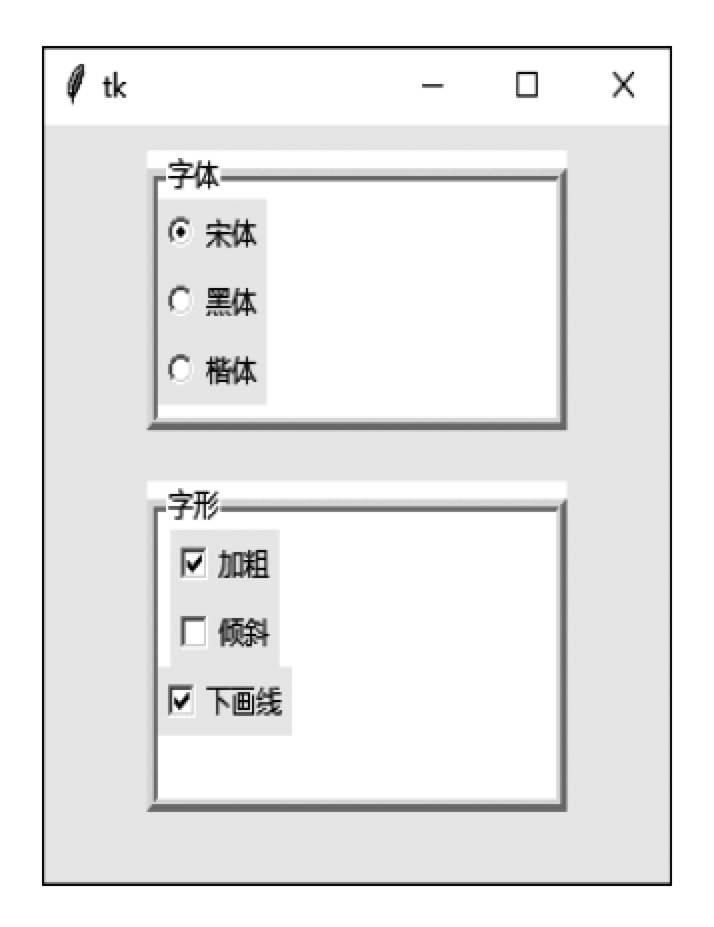


图 10.12　标签框架实例

(6) 列表框(Listbox)组件

列表框组件包含一个选项列表,用户可以从列表中选择一个或多个列表项。创建列表框组件的格式与创建其他组件类似,具体如下:

```
<列表框组件名>=tkinter.Listbox(<父组件>,[参数列表],…)
```

列表框的常用属性及方法见表 10.11 及表 10.12 所示。

表 10.11　列表框组件的常用属性

属　　性	说　　明
selectmode	选择模式: SINGLE——单选 BROWSE——单选,拖动鼠标或通过方向键可以选择多行 MULTIPLE——多选 EXTENDED——多选,但需要同时按住 Shift 键或 Ctrl 键或拖曳鼠标实现
xscrollcommand	水平滚动条联系变量
yscrollcommand	垂直滚动条联系变量
listvariable	列表框内容的 StringVar 变量,可以用.get()方法得到列表框内容的字符串

表 10.12　列表框组件的常用方法

方　　法	说　　明
Insert(index, * elements)	插入新的列表项
curselection()	获得选中列表项索引值的元组
delete(first,last=None)	删除列表项
get(first,last=None)	获得指定列表项内容的元组

列表框还可以有水平或垂直滚动条,通过将列表框与滚动条组件联系实现。这方面的内容在此不做深入介绍,有兴趣的读者可以进一步阅读相关书籍。下面的示例为创建一个列表框,并通过回调函数将列表框中选中的内容在标签组件中显示。

【例 10.13】 列表框组件的使用。

程序代码如下:

```
#example10.13
from tkinter import *
root=Tk()
```

```
def printLabel(x):
    Label(root,text=lb.get(lb.curselection()),\
        font=('黑体',48),bg='red').grid(row=0,column=0)

lb=Listbox(root)
lb.bind('<Button-1>',printLabel)
lb.grid(row=1,column=0)
for i in range(10):
lb.insert(END,str(i*100))
root.mainloop()
```

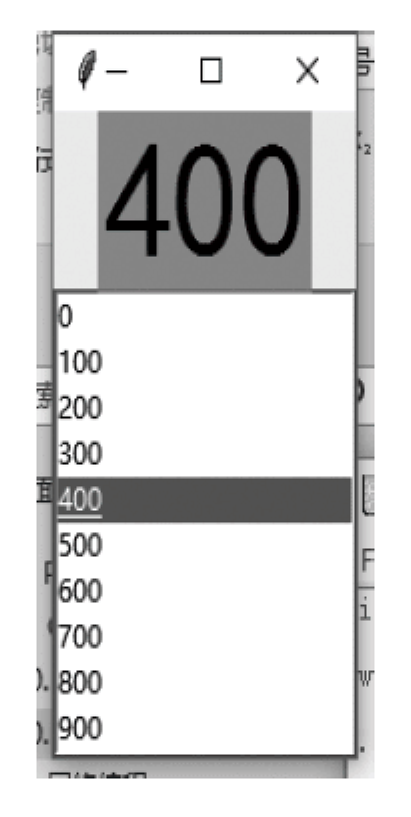


图 10.13 列表框组件实例

程序运行结果如图 10.13 所示。

(7) 菜单(Menu)组件

tkinter 的菜单有两种：下拉式菜单和快捷菜单。

下拉式菜单由菜单栏和一组弹出式子菜单组成。菜单栏由若干主菜单项组成，主菜单项横向排列成一行，也称为顶层菜单，每个主菜单项对应一个分类弹出式子菜单，子菜单中的每一项称为菜单项，菜单项可以是独立的菜单命令、分割线或者下一级子菜单。

快捷菜单是单击鼠标右键时出现的菜单，可以增加程序的可操作性、提高程序效率。

创建下拉式菜单的格式如下：

```
<菜单组件名>=tkinter.Menu(<父组件>,[参数列表],…)
```

根据父组件参数的不同，可以创建菜单栏主菜单(父组件是顶层窗口)和弹出式子菜单(父组件是主菜单项)。

菜单的绝大多数属性与前述组件相同，应用菜单组件创建菜单的过程主要使用菜单的方法实现，常见的菜单方法如表 10.13 所示。

以上方法，常用的参数主要有：label(菜单项名称)、command(单击菜单项时调用的方法)、accelerator(快捷键)和 underline(菜单项添加下画线)。

菜单栏中的多选按钮和单选按钮都是一组菜单项，选中某项菜单项后，菜单项左边有选中标志(√)，单选按钮和多选按钮菜单项的区别是单选按钮组内只能同时选中一项，多选按钮菜单项可以同时选中多项。下面的示例为创建一个包含子菜单的顶层菜单，模仿 PythonIDLE 的 File 菜单项设计。

表 10.13 菜单的常用方法

方　法	说　明
add_command(option, …)	添加菜单项,menu 参数指定顶层菜单
add_cascade(option, …)	添加级联菜单,menu 参数指定被级联菜单
add_checkbutton(option, …)	添加多选按钮菜单项
add_radiobutton(option, …)	添加单选按钮菜单项
add_separator()	添加分隔线
insert_command(index,option, …)	插入菜单项
insert_cascade(index,option, …)	插入级联菜单
insert_checkbutton(index,option, …)	插入多选按钮菜单项
insert_radiobutton(index,option, …)	插入单选按钮菜单项
insert_separator(index)	插入分隔线
post(x,y)	弹出菜单的位置坐标

【例 10.14】 菜单组件简单应用。

程序代码如下：

```
#example10.14 菜单组件
import tkinter
top=tkinter.Tk()
top.geometry('300x450+100+100')
top.wm_title('Python Shell')
main_m=tkinter.Menu(top)                              #创建主菜单
item=tkinter.Menu(main_m, tearoff=0)                  #创建子菜单
for i in ['New File','Open','Open Module','Recent Files',\
        'Class Browser','Path Browser']:
    item.add_checkbutton(label=i)                     #添加菜单项
item.add_separator()                                  #添加分隔线
for i in['Save','Save as','Save Copy AS']:
    item.add_radiobutton(label=i)                     #添加菜单项
item.add_separator()                                  #添加分隔线
item.add_radiobutton(label='Print Window',accelerator='Ctrl+P')
item.add_separator()                                  #添加分隔线
item.add_radiobutton(label='Close',accelerator='Alt+F4')
```

```
item.add_radiobutton(label='Exit',accelerator='Ctrl+Q')
main_m.add_cascade(label='File',menu=item)            #指定顶层菜单
top['menu']=main_m
top.mainloop()
```

程序运行结果如图 10.14 所示。

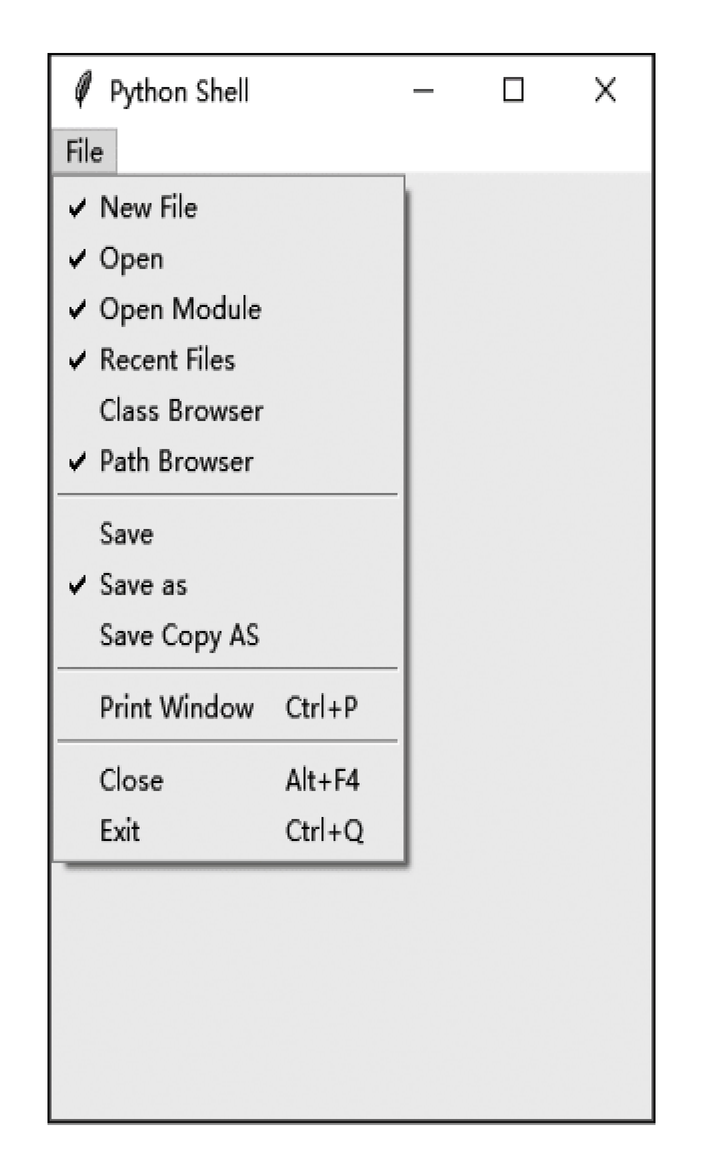


图 10.14 菜单实例

快捷菜单是在顶层窗口单击鼠标右键后产生的,因此快捷菜单没有顶层菜单,只有菜单项。在窗口中单击右键的位置由 post()方法记住,快捷菜单在 post()方法的位置(x 和 y 坐标)弹出。下面的示例将上例进行简单修改,由下拉式菜单变为快捷菜单。

【例 10.15】 快捷菜单简单实例。

程序代码如下:

```
#example10.15 快捷菜单
import tkinter
top=tkinter.Tk()
top.geometry('300x450+100+100')
top.wm_title('Python Shell')

def post_menu(p):
    item.post(p.x, p.y)
```

```
item=tkinter.Menu(top, tearoff=0)                    #创建快捷菜单
for i in ['New File','Open','Open Module','Recent Files',\
            'Class Browser','Path Browser']:
      item.add_checkbutton(label=i)                  #添加菜单项
item.add_separator()                                 #添加分隔线
for i in['Save','Save as','Save Copy AS']:
      item.add_command(label=i)                      #添加菜单项
item.add_separator()                                 #添加分隔线
item.add_command(label='Print Window',accelerator='Ctrl+P')
item.add_separator()                                 #添加分隔线
item.add_command(label='Close',accelerator='Alt+F4')
item.add_command(label='Exit',accelerator='Ctrl+Q')
top.bind('<Button-3>',post_menu)                     #回调函数,弹出快捷菜单
top.mainloop()
```

程序运行结果如图 10.15 所示。

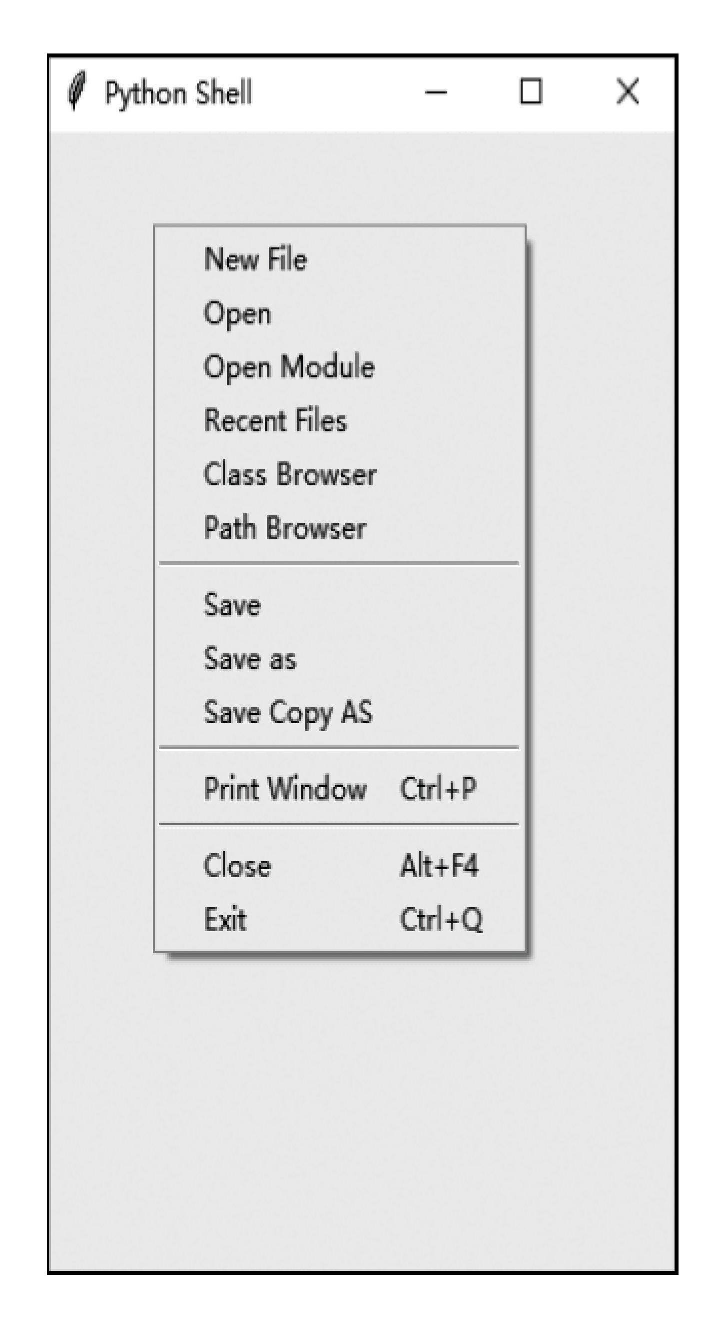
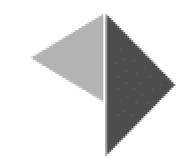


图 10.15　快捷菜单实例

tkinter 还有画布(Canvas)组件、滚动条组件(Scrollbar)、进度条(Scale)组件等其他组件,由于篇幅所限,不做一一介绍,感兴趣的读者可以参考其他相关书籍。



10.2 网络编程

Python 的标准库中的有些库封装了常用的网络协议，如 socket、urllib 和 urllib2 等，利用这些库可以实现很多网络功能。此外，Python 还提供 wget、scrapy、requests 等大量第三方模块实现对网页内容进行捕获和处理，可以快速开发网页爬虫等应用。因此，Python 是一个非常强大的网络编程工具。

10.2.1 Socket 编程

Socket 是计算机之间进行网络通信的一套程序接口，也是计算机进程间通信的一种方式，它可以实现不同主机进程间的通信。网络上各种服务大多是基于 Socket 完成通信的，如浏览网页、QQ 聊天、收发 E-mail 等。Python 标准库的 socket 模块支持 Socket 接口访问，利用它可以极大地提高网络程序的开发效率。Socket 模块包括两个部分：服务端和客户端，服务端负责监听端口号，等待客户端发送消息；客户端在需要发送信息时，连接服务端，将信息发送出去。

TCP 和 UDP 是网络体系结构传输层最重要的两个协议。TCP 协议负责在两台计算机之间建立可靠连接，保证数据包按顺序到达。因此，TCP 协议适合对准确性要求较高的场合，如文件传输、电子邮件等。UDP 是面向无连接的协议，使用 UDP 协议传输数据，不需要事先建立连接，只需要知道对方的 IP 地址和端口号即可，但是 UDP 协议不保证数据能准确送达。虽然 UDP 传输数据不可靠，但它的传输速度快，因此对于可靠性要求不高的场合，可以使用 UDP 协议，如网络语音通信、视频点播等。

1. TCP 编程

TCP 是一种面向连接的传输层协议，TCP Socket 是基于一种 C/S 的编程模型，服务端监听客户端的连接请求，一旦建立连接即可进行传输数据。

(1) 客户端编程

Socket 模块客户端编程主要分为以下 5 个步骤。

① 创建 socket

利用 socket 模块的 socket 函数实现，具体格式如下：

```
s=socket.socket([family,[type,[proto]]])
```

其中，family 可以是 socket. AF_INEF(IPv4)、socket. AF_INET6(IPv6)或者 AF_

UNIX(同一台机器进程间的通信),type 为套接字类型,可以是 SOCKET_STREAM(流式套接字,用于 TCP 协议)或者 SOCKET_DGRAM(数据报套接字,用于 UDP 协议)。

② 连接服务器

使用连接函数 connect 连接到远程服务器 IP 的某个特定端口上。格式如下:

```
s.connect((remote_ip , port))
```

③ 发送数据

连接服务器成功后,可以向服务器发送一些数据。如:

```
s.sendall(b''GET/HTTP/1.1\r\nHost: www.baidu.com\r\nConnection: close\r\n\r\n')
```

④ 接收数据

发送完数据之后,客户端需要接收服务器的响应,使用函数 recv()实现。如:

```
reply=s.recv(4096)
```

⑤ 关闭 socket

接收完服务器数据后,可以将该 socket 关闭,结束这次通信。如:

```
s.close()
```

下面是一个客户端程序的实例。

【例 10.16】 TCP 客户端编程实。

程序代码如下:

```
#example10.16 tep-client
import socket                              #导入 socket 库
#创建一个 socket:
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('www.baidu.com', 80))           #建立与远程主机的连接
#发送数据,请求网页内容
s.send(b'GET/HTTP/1.1\r\nHost:www.baidu.com\r\nConnection: close\r\n\r\n')
buf=[]
while True:                                #接收数据
    d=s.recv(1024)                         #每次最多接收 1K 字节
```

```
    if d:
        buf.append(d)
    else:
        break
data=b''.join(buf)
s.close()                                   #关闭连接
header, html=data.split((b'\r\n\r\n'), 1)
print(header.decode('utf-8'))
with open('baidu.html', 'wb') as b:         #把接收到的数据写入文件
    b.write(html)
```

(2) 服务器编程

服务器编程主要有以下 5 方面的工作。

① 打开 socket

② 绑定监听地址以及端口

可以使用 127.0.0.1 绑定本机地址,但客户端必须在本机运行才能建立连接。标准服务的端口号都需要预先指定,如 Web 服务器 HTTP 端口一般为 80,FTP 端口为 21、Telnet 端口为 23。函数 bind()可以用来将 socket 绑定到特定的地址和端口上。

③ 监听连接

调用 listen()函数监听端口,该函数带有参数,用来控制连接的最大数量。

④ 建立连接

当有客户端向服务器发送连接请求时,服务器通过一个永久循环接受连接,accept()会等待并返回一个客户端的连接。

⑤ 接收/发送数据

服务器端与客户端实现收发数据操作。

下面的示例是一个简单的服务器端程序。

【例 10.17】 TCP 服务器端编程实例。

程序代码如下:

```
#example10.17 tcp-service
import socket
import sys
HOST='127.0.0.1'                  #本机地址
PORT=5000                         #任意非特定端口
```

```
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print('Socket 创建成功')
s.bind((HOST, PORT))                    #绑定端口
print('Socket 绑定端口成功')
s.listen(5)
print('Socket 开始监听')
while True:                             #保持与客户端的持续连接
    conn, addr=s.accept()               #接受新连接
    print('Connected with '+ddr[0]+':'+str(addr[1]))
    #创建新线程处理 TCP 连接:
    t=threading.Thread(tcplink, args=(sock, addr))
    t.start()

def tcplink(sock, addr):
    print('请输入新的连接 %s:%s…' %addr)
    sock.send(b'welcome!')
    while True:
        data=sock.recv(1024)
        time.sleep(1)
        if not data or data.decode('utf-8')=='exit':
            break
sock.send(('Hello, %s!' %data.decode('utf-8')).encode('utf-8'))
sock.close()
print('Connection from %s:%s closed.' %addr)
```

说明：在该例的程序代码中，建立连接后，服务器首先发一条欢迎消息，然后等待客户端数据，客户端用户输入数据后，服务器端将数据加上“Hello”并返回给客户端，如果客户端发送 exit 字符串，则关闭连接。

下例中的客户端程序可以用来对上述服务器程序进行测试。

【例 10.18】 TCP 客户端编程实例。

程序代码如下：

```
#example10.18 tcp-client
import socket
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('127.0.0.1', 5000))                    #建立连接
```

```
print(s.recv(1024).decode('utf-8'))            #接收欢迎消息
for data in [b'Michael', b'Tracy', b'Sarah']:
    s.send(data)                               #发送数据
    print(s.recv(1024).decode('utf-8'))
s.send(b'exit')
s.close()
```

2. UDP 编程

和 TCP 类似,UDP 编程也需要将通信双方分为客户端和服务器。服务器端绑定需要端口,但不需要用 listen()进行监听,而是直接接收来自任何客户端的数据。客户端不需要调用 connect()与服务器进行连接,直接将数据发送给服务器。下面的示例简单地表达了这样的过程。

【例 10.19】 UDP 编程实例。

服务器端代码:

```
#-*-coding: utf-8-*-
#example10.19-udp-server
import socket
#创建一个 socket,SOCK_DGRAM表示 UDP
s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('127.0.0.1', 10021))                    #绑定 IP 地址及端口
print('UDP 连接')
while True:
#获得数据和客户端的地址与端口,一次最多接收 1024 字节
    data, addr=s.recvfrom(1024)
    print('收到数据 %s:%s.' %addr)
    s.sendto(data.decode('utf-8').upper().encode(), addr)          #数据大写送回客户端
#不关闭 socket
```

客户端代码:

```
#-*-coding: utf-8-*-
#example10.19-udp-client
import socket
s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
addr=('127.0.0.1', 10021)                      #服务器端地址
while True:
```

```
        data=input('请输入要处理的数据:')              #获得数据
        if not data or data=='quit':
            break
        s.sendto(data.encode(), addr)                  #发送到服务端
        recvdata, addr=s.recvfrom(1024)                #接收服务器端发来的数据
        print(recvdata.decode('utf-8'))                #解码打印
    s.close()                                          #关闭 socket
```

10.2.2　Python 网络爬虫

网络爬虫是一种按照一定的规则自动地捕获万维网信息的程序或者脚本。爬虫程序通常从网站的某一个页面(通常是首页)开始,读取网页的内容,找到在网页中的其他链接地址,然后通过这些链接地址寻找下一个网页,然后一直循环下去,直到把这个网站所有的网页都捕获完为止。

网络爬虫一般分为两个步骤。

(1) 获取网页内容。

(2) 对获取的网页内容进行分析处理。

在 Python 中,有很多库可以实现网络爬虫功能,如 urlib、urlib2、urlib3、requests 等。在 Python 3.X 中,目前使用较多的是 requests 库和 beautifulsoup4 库。requests 主要用来获取网页内容,beautifulsoup4 库用来分析网页数据。这两个第三方库需要单独下载安装,具体方法可以参考本书第 6 章。下面根据网络爬虫的两个步骤简要介绍爬虫过程。

1. 获取网页内容——requests 库

Urllib 库和 requests 库都是比较好的捕获网页的库。Urllib 是 Python 的标准库,提供了 urllib.request、urllib.response、urllib.parse 和 urllib.error 四个模块。requests 库是非常优秀的处理 HTTP 请求的第三方库,它基于 urllib,但比 urllib 更加方便,功能更强大,完全满足 HTTP 测试的需求。本书将以 requests 库为例,介绍网络爬虫的网页捕获功能。

可以采用 pip 指令安装 requests 库:pip install requests。

下面的代码用 requests 库实现读取并显示网页的内容。

```
>>>import requests
>>>response=requests.get('http://www.baidu.com')
>>>type(response)
<class 'requests.models.Response'>
```

最基本的请求可以直接使用 get()方法，如果需要参数，则可以利用 params。如在向服务器发起 get 请求时，问号(?)后面的字符串会被解析为查询字符串，格式一般是?key1=value1&key2=value2，如果手工构建 URL，那么数据会以键/值对的形式置于 URL 中，连接在一个问号的后面。例如，http://www.baidu.com/get?key=val。Requests 允许使用 params 关键字参数，以一个字典提供这些参数。例如，想传递 key1=value1 和 key2=value2 到 http://www.baidu.com，那么可以使用如下代码：

```
>>>payload={'key1': 'value1', 'key2': 'value2'}
>>>r=requests.get('http://www.baidu.com',params=payload)
>>>print(r.url)
```

通过打印输出的该 URL 已被正确编码如下：

```
http://www.baidu.com/? key1=value1&key2=value2
```

除了 get 外，requests 还支持其他 HTTP 请求，例如：

```
>>>r=requests.post('http://www.baidu.com')
>>>r=requests.put('http://www.baidu.com')
>>>r=requests.delete('http://www.baidu.com')
>>>r=requests.head('http://www.baidu.com')
>>>r=requests.options('http://www.baidu.com')
```

单纯从获取网页的角度来讲，requests 的 get()方法已经足以满足各种要求，get()方法的返回值会保存为一个 Response 对象，Response 作为服务器对 get()响应的结果，具有自己的属性和方法，网络爬虫的后续操作主要通过调用 Response 对象的属性和方法实现相关功能。

Response 对象的主要属性如下。

(1) status_code 属性：HTTP 请求的返回状态，200 表示连接成功，404 表示连接失败。

(2) text 属性：HTTP 响应的字符串内容，会自动根据响应头部的字符编码进行解码。

(3) encoding 属性：HTTP 响应的编码方式。

(4) content 属性：HTTP 响应的字节方式。

(5) headers 属性：以字典对象存储服务器响应头，字典键不区分大小写，若键不存在，则返回 None。

下面的代码分别验证以上属性的功能。

```
>>>r=requests.get('http://www.baidu.com')
>>>r.status_code
200
>>>r.text
```

输出略。

```
>>>r.encoding
'ISO-8859-1'
>>>r.content
```

输出略。

```
>>>r.headers
{'Server': 'bfe/1.0.8.18', 'Date': 'Wed, 10 May 2017 01:08:25 GMT', 'Content-Type': 'text/
html', 'Last-Modified': 'Mon, 23 Jan 2017 13:27:27 GMT', 'Transfer-Encoding': 'chunked', '
Connection': 'Keep-Alive', 'Cache-Control': 'private, no-cache, no-store, proxy-
revalidate, no-transform', 'Pragma': 'no-cache', 'Set-Cookie': 'BDORZ=27315; max-age=
86400; domain=.baidu.com; path=/', 'Content-Encoding': 'gzip'}
```

Response 的方法主要有两个。

(1) json()方法：内置的 JSON 解码器，解析 JSON 数据。

(2) raise_for_status()方法：如果 get 请求失败(非 200 响应)，则抛出异常。

这两个方法的使用请参考下面的示例。

【例 10.20】 raise_for_status()方法和 json()方法的应用示例。

程序代码如下：

```
#example10.20
import requests
URL='http://ip.taobao.com/service/getIpInfo.php'          #淘宝 IP 地址库 API
try:
    r=requests.get(URL, params={'ip': '8.8.8.8'}, timeout=1)
    r.raise_for_status()                        #如果响应状态码不是 200,则主动抛出异常
except requests.RequestException as e:
    print(e)
else:
    result=r.json()
print(type(result), result, sep='\n')
```

程序运行结果如下：

```
<class 'dict'>
{'code': 0, 'data': {'country': '美国', 'country_id': 'US', 'area': '', 'area_id':
'', 'region': '', 'region_id': '', 'city': '', 'city_id': '', 'county': '', 'county_id': '
', 'isp': '', 'isp_id': '', 'ip': '8.8.8.8'}}
```

2. 分析获取数据——beautifulsoup4 库

使用 requests 库获取 HTML 网页内容后，需要解析 HTML 页面，这就需要用到能从 HTML 页面提取有用数据的函数库。beautifulsoup4 库是一个非常优秀的 Python 扩展库，不仅可以解析 HTML 页面内容，还可以读取 XML 文件内容。beautifulsoup4 库也称为 beautifulsoup 库或 bs4 库，可以使用 pip install beautifulsoup4 进行安装，安装后使用 from bs4 import BeautifulSoup4 进行导入。

熟悉 HTML 的人都知道，由 requests 获取的 HTML 页面一般都比较复杂，包含大量的格式信息，直接解析比较困难，这是由 HTML 的语法结构决定的。beautifulsoup4 库提供一些直接处理 HTML 页面的函数（方法），可以方便快捷地解析页面元素。beautifulsoup4 库将解析的每一个 HTML 页面当作一个对象，通过调用页面对象的属性和方法实现对页面的解析。HTML 页面中的每一个标签（Tag）都是 beautifulsoup4 创建的页面对象的一个属性，如<body>、<head>等。下面的代码列举了调用页面对象属性的方法：

```
>>>import requests
>>>from bs4 import BeautifulSoup              #导入 bs4 库中的 BeautifulSoup 类
>>>r=requests.get('http://www.baidu.com')
>>>r.encoding='utf-8'
>>>soup=BeautifulSoup(r.text)
>>>type(soup)#soup 是一个 BeautifulSoup 类的一个对象
<class 'bs4.BeautifulSoup'>
>>>soup.head
<head><meta content="text/html;charset=utf-8" http-equiv="content-type"><meta
content="IE=Edge" http-equiv="X-UA-Compatible"><meta content="always" name="
referrer"><link href="http://s1.bdstatic.com/r/www/cache/bdorz/baidu.min.css" rel="
stylesheet" type="text/css"><title>百度一下，你就知道</title></link></meta></meta
></meta></head>
>>>soup.p
```

```
<p id="lh"><a href="http://home.baidu.com">关于百度</a><a href="http://ir.baidu.
com">About Baidu</a></p>
>>>soup.a
<a class="mnav" href="http://news.baidu.com" name="tj_trnews">新闻</a>
>>>soup.title
<title>百度一下,你就知道</title>
```

同时,soup 对象的每一个标签也是一个对象,称为 Tag 对象。Tag 对象有四个常用属性,具体如下。

(1) name 属性:标签的名字。

(2) attrs 属性:以字典形式返回原页面所有标签的所有属性。

(3) contents 属性:以列表形式返回该标签下所有标签的内容。

(4) string 属性:以字符串形式返回标签包含的文本内容,如果标签内嵌套一层标签,则返回最内层标签的内容;如果标签内嵌套多层标签,则返回 None。

下面的代码展示了以上属性的用法:

```
>>>soup.title.name
'title'
>>>soup.title.string
'百度一下,你就知道'
>>>soup.title.parent.name
'link'
>>>soup.p
<p id="lh"><a href="http://home.baidu.com">关于百度</a><a href="http://ir.baidu.
com">About Baidu</a></p>
>>>soup.a#返回第一个超链接
<a class="mnav" href="http://news.baidu.com" name="tj_trnews">新闻</a>
>>>soup.a.attrs
{'href': 'http://news.baidu.com', 'name': 'tj_trnews', 'class': ['mnav']}
>>>soup.a.string
'新闻'
>>>soup.p.contents
[' ',<a href="http://home.baidu.com">关于百度</a>, ' ',<a href="http://ir.baidu.com"
>About Baidu</a>, ' ']
>>>soup.find_all('a')              #查找所有的超链接
```

```
[<a class="mnav" href="http://news.baidu.com" name="tj_trnews">新闻</a>,
<a class="mnav"href="http://www.hao123.com" name="tj_trhao123">hao123</a>,<a class=
"mnav" href="http://map.baidu.com" name="tj_trmap">地图</a>,<a class="mnav" href="
http://v.baidu.com" name="tj_trvideo">视频</a>,<a class="mnav" href="http://tieba.
baidu.com" name="tj_trtieba">贴吧</a>,<a class="lb" href="http://www.baidu.com/
bdorz/login.gif? login&amp;tpl=mn&amp;u=http%3A%2F%2Fwww.baidu.com%2f%3fbdorz_
come%3d1" name="tj_login">登录</a>,<a class="bri" href="//www.baidu.com/more/" name
="tj_briicon" style="display:block;">更多产品</a>,<a href="http://home.baidu.com">
关于百度</a>,<a href="http://ir.baidu.com">About Baidu</a>,<a href="http://www.
baidu.com/duty/">使用百度前必读</a>,<a class="cp-feedback" href="http://jianyi.
baidu.com/">意见反馈</a>]
```

关于 beautifulsoup4 库的强大功能,更加详细完整的信息请参考 http://www.crummy.com/software/BeautifulSoup4/bs4/doc/。

3. 网络爬虫示例

下面通过两个简单的网络爬虫示例,进一步了解网络爬虫的基本步骤,理解爬虫的主要功能。

【例 10.21】 抓取淘宝网首页的中文产品类别。

程序代码如下:

```
#-*-coding:utf-8-*-
#example10.21
import requests
from bs4 import BeautifulSoup
r=requests.get('http://www.taobao.com')
r.encoding='utf-8'
soup=BeautifulSoup(r.text, "html.parser")
for list in soup.find_all('a'):
    if not list.string==None:
        print(list.string)
```

运行程序,部分结果如图 10.16 所示。

说明:

(1) 利用 requests 库抓取网页内容,利用 beautifulsoup4 库分析网页中的数据。

(2) 分析获取的 HTML 页面数据格式,发现产品类别信息被封装在<a>之间的结构中,因此,要获得产品类别信息,首先需要找到<a>标签,获取其中的数据并输出。

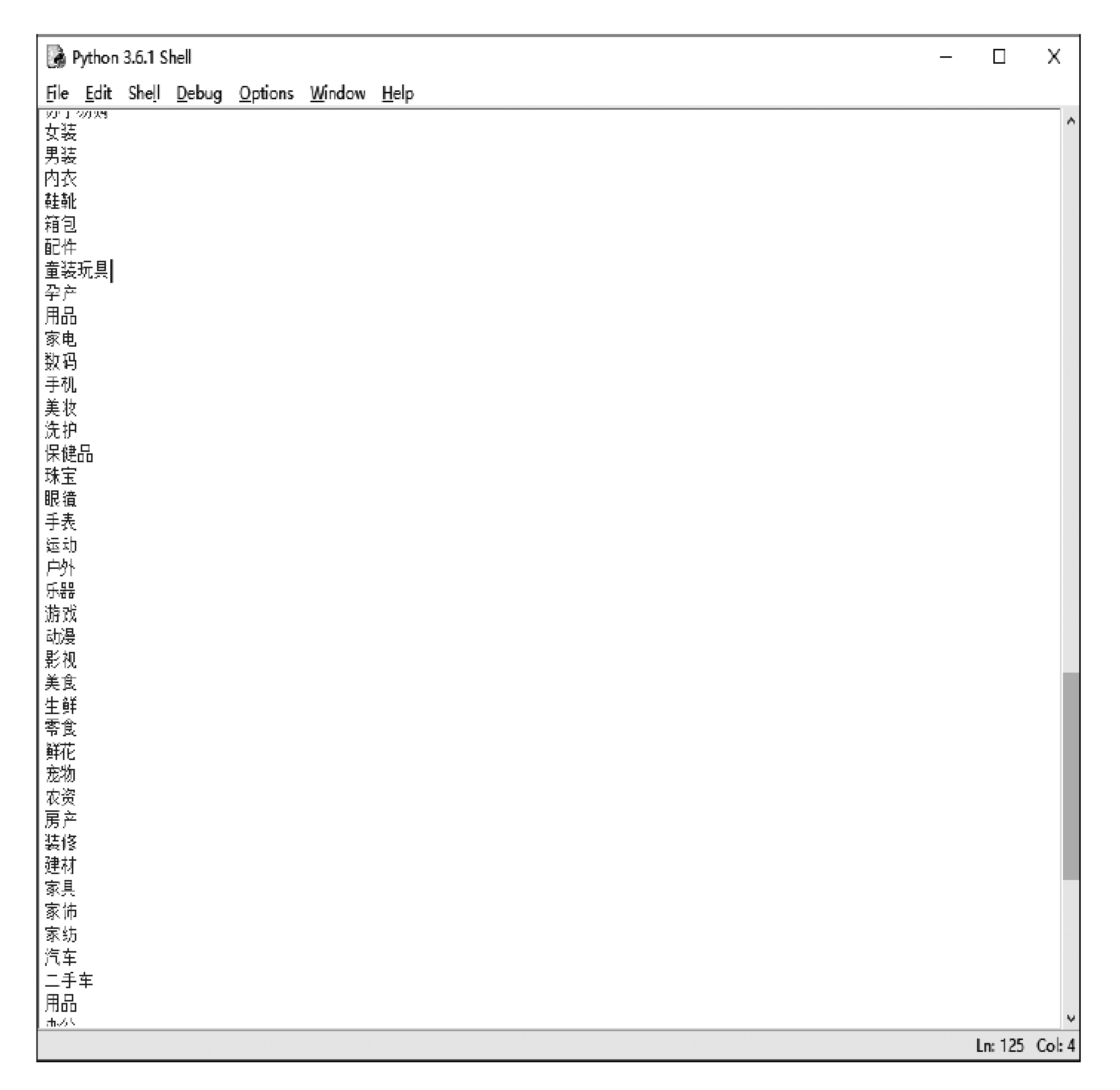


图 10.16 抓取淘宝网首页的中文产品类别

【例 10.22】 抓取糗事百科热门段子。

程序代码如下：

```
#-*-coding: utf-8-*-
#example10.22
import requests
from bs4 import BeautifulSoup
content=requests.get('http://www.qiushibaike.com').content
soup=BeautifulSoup(content, 'html.parser')
for div in soup.find_all('div', {'class' : 'content'}):
    print(div.text.strip())
```

程序运行结果如图 10.17 所示。

说明：分析获取的 HTML 页面，发现网页的所有文本内容都被封装在<div class="content"></div>标签内。

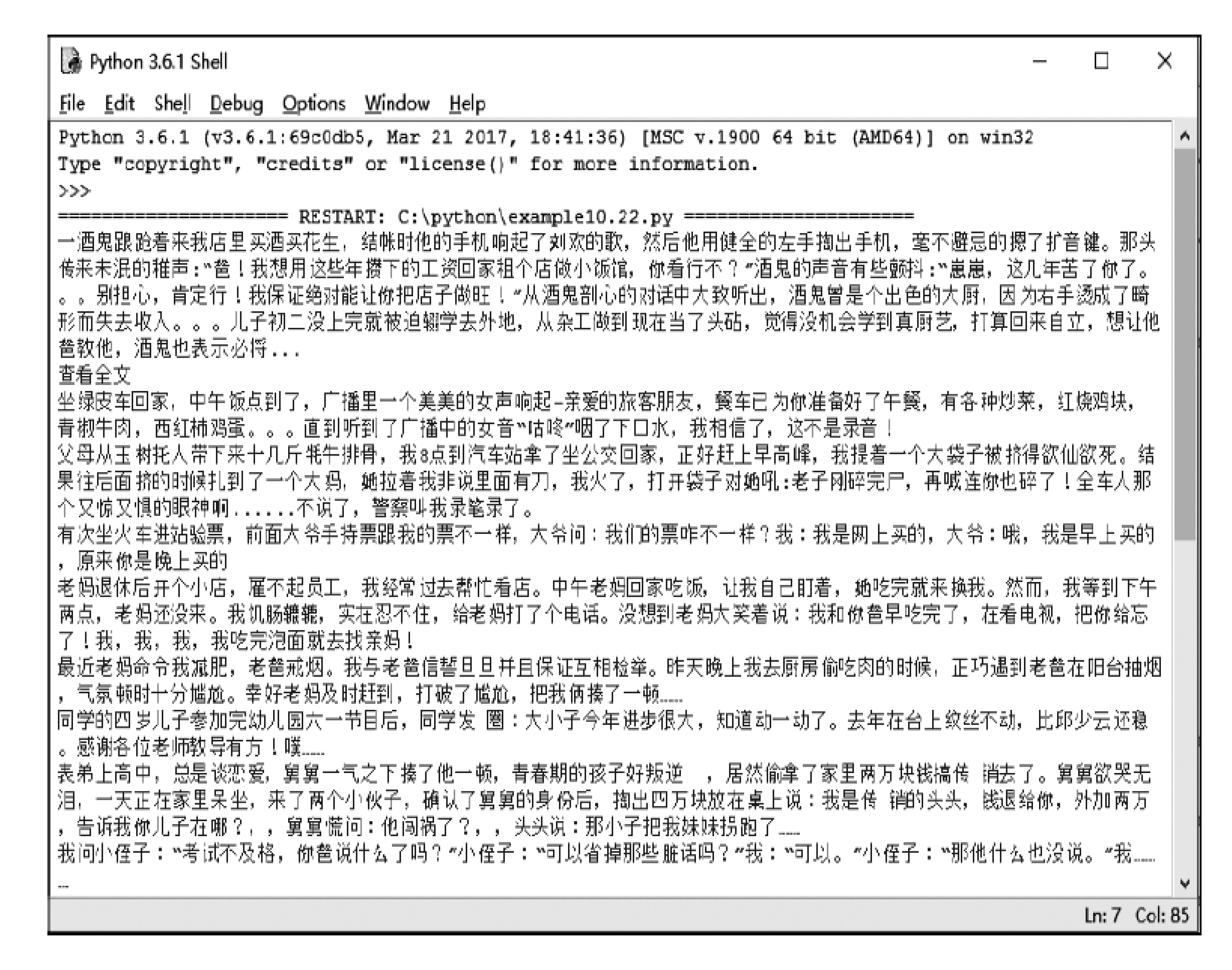


图 10.17　爬取糗事百科热门段子



10.3　数据库编程

数据库技术为数据的共享、查询修改等操作提供了有力的技术支撑，在各行各业的应用程序中被广泛采用，如大型网站、管理信息系统、邮箱系统等。同时，大数据时代的到来进一步促进了数据库技术的发展。在 Python 语言中，有很多方法可以提供数据库存储功能。对于 MySQL、Access、Oracle、Sybase、Microsoft SQL Server、SQLite 等关系型数据库，Python 都提供了 ODBC 接口。Python 提供了一个标准接口用于访问关系型的数据库，即 Python DB Application Programming Interface(Py-DBAPI)。每种数据库 API 都需要有一个 Py-DBAPI 封装的实现，而几乎所有的数据库接口都有对应的实现方法，同时，Python 语言还内置了用于存储和获取数据的工具。本节重点介绍 Python 标准库自带的 SQLite 接口，实现对 SQLite 数据库的操作。

10.3.1　SQLite 数据库简介

SQLite 数据库是一款非常小巧的嵌入式开源数据库软件，没有独立的维护进程，所有的维护都来自于程序本身。它是遵守 ACID 的关联式数据库管理系统，它的设计目标

是嵌入式的，而且目前已经在很多嵌入式产品中使用，它占用的资源非常少，在嵌入式设备中，可能只需要几百 KB 的内存。它能够支持 Windows/Linux/UNIX 等主流的操作系统，同时能够与很多程序语言相结合，如 Tcl、C＃、PHP、Java 等，还有 ODBC 接口，同样比起 MySQL、PostgreSQL 这两款开源的世界著名数据库管理系统而言，它的处理速度比它们都快。SQLite 的第一个 Alpha 版本诞生于 2000 年 5 月，目前被广泛使用的版本是 SQLite 3。

SQLite 引擎不是独立于程序之外的独立进程，而是连接到程序中成为它的一个主要部分。主要的通信协议是在编程语言内直接通过 API 调用，这在消耗总量、延迟时间和整体简单性上均有积极的作用。整个数据库(定义、表、索引和数据本身)都在宿主主机上存储在一个单一的文件中。

10.3.2 Python 操作 SQLite 数据库

SQLite 是 Python 自带的一个轻量级关系型数据库，Python 标准库中的 sqlite3 提供该数据库的接口，不需要单独安装就可以使用，支持使用 SQL 语句访问数据库，使用时通过调用 sqlite3 模块与 Python 进行集成。SQLite 作为后端数据库，可以搭配 Python 建立网站，或者制作有数据存储需求的工具。为了使用 sqlite3 模块，必须首先创建一个表示数据库的连接对象，然后可以有选择地创建游标对象，这将有助于执行所有的 SQL 语句。

1. 连接数据库

要操作关系数据库，首先需要连接到数据库，创建一个与数据库关联的 Connection 对象，例如下面的代码：

```
>>>import sqlite3#导入 sqlite3模块
>>>conn=sqlite3.connect('test.db')#创建数据库连接对象 conn
```

首先导入 sqlite3 模块，然后创建数据库 test.db，如果 test.db 已经存在，则直接打开这个数据库，同时创建一个与 test.db 关联的数据库连接对象 conn。

Connection 对象是 sqlite3 模块中最重要的类，主要方法如下。

(1) cursor()：打开数据库连接对象的游标。

```
c=conn.cursor()
```

(2) commit()：提交当前事物，保存数据。若不提交，则不保存数据，数据库中为上次调用 commit()方法之后的数据。

```
conn.commit():
```

(3) rollback():撤销当前事物,恢复到上次调用 commit()方法后的数据状态。

```
conn.rollback()
```

(4) close():关闭数据库连接。

```
conn.close()
```

2. 使用游标查询数据库

创建数据库连接对象 conn 连接到数据库后,需要打开游标,称之为 Cursor,通过 Cursor 执行 SQL 语句实现对数据库表的查询、插入、修改、删除等操作。在 sqlite3 中,所有 SQL 语句的执行都要在游标对象的参与下完成,Cursor 对象的主要方法如下。

(1) execute(sql[,parametres]):执行一条 SQL 语句,例如,上面的两条语句执行后,继续执行下面的语句。

```
c.execute('''create table cata(id int primary key,pid int,name varchar(10) UNIQUE,
nickname text)'''
```

(2) executemany(sql[,parametres]):执行多条 SQL 语句,对于所有给定参数执行同一条 SQL 语句,一般参数是一个序列。

(3) fetchone():从结果中取出一条记录。

(4) fetchmany():从结果中取出多条记录。

(5) fetchall():从结果中取出所有记录。

(6) scroll():游标滚动。

下面的程序实现创建数据库和数据表、插入记录以及查询记录的过程。

【例 10.23】 sqlite3 的基本操作。

程序代码如下:

```
#example10.23
import sqlite3                                  #导入 sqlite3 模块
conn=sqlite3.connect('school.db')               #连接数据库 school.db
cursor=conn.cursor()                            #打开游标
print('数据库已打开!')
```

```
#创建表
cursor.execute('''create table student(ID int primary key not null,
                        Name text not null,Age int not null,
                        Score real,Address char(50))''')
print('创建表成功!')
#插入记录
cursor.execute("insert into student values(1001,'张晓山',20,589,'辽宁沈阳')")
cursor.execute("insert into student values(1002,'孙晓宇',19,568,'山东青岛')")
cursor.execute("insert into student values(1003,'宋健',20,590,'吉林四平')")
cursor.execute("insert into student values(1004,'潘恩东',21,526,'河北承德')")
cursor.execute("insert into student values(1005,'赵子瑜',20,601,'江苏苏州')")
conn.commit()                                       #提交当前事物,保存数据
for row in cursor.execute('select * from student'): #查询表中内容
    print(row)                                      #输出表中内容
conn.close()                                        #关闭数据库连接
```

程序运行结果如下：

```
>>>
================RESTART:C:\python\example10.23.py================
数据库已打开!
创建表成功!
(1001, '张晓山', 20, 589.0, '辽宁沈阳')
(1002, '孙晓宇', 19, 568.0, '山东青岛')
(1003, '宋健', 20, 590.0, '吉林四平')
(1004, '潘恩东', 21, 526.0, '河北承德')
(1005, '赵子瑜', 20, 601.0, '江苏苏州')
>>>
```

习题 10

一、填空题

1. Python 用来访问和操作内置数据库 SQLite 的标准库是________。
2. tkinter 模块的布局方法主要有 pack()方法、grid()方法和________。

二、判断题

1. 在 GUI 设计中，复选框往往用来实现非互斥多选的功能，多个复选框之间的选择互不影响。 （ ）

2. 在 GUI 设计中，单选按钮用来实现用户在多个选项中的互斥选择，在同一组内的多个选项中只能选择一个，当选择发生变化之后，之前选中的选项自动失效。 （ ）

3. Python 只能使用内置数据库 SQLite，无法访问 MS SQLServer、ACCESS 或 Oracle、MySQL 等数据库。 （ ）

4. top 是类 Tk 的实例，命令行 top.mainloop()可以实现 GUI 程序的主事件循环。 （ ）

5. 按钮组件的 command 属性可以指定某个函数，实现用户单击按钮事件发生时所要求的功能。 （ ）

6. 用户在输入框组件中输入的信息可以通过输入框组件的 get()函数获得。（ ）

7. 在下拉式菜单中，可以定义选择按钮或者单选按钮。 （ ）

8. UDP 协议负责在两台计算机之间建立可靠连接，保证数据包按顺序到达，一般用于对可靠性要求较高的场合。 （ ）

9. 在目前实现网络爬虫功能的两个主要第三方库中，requests 库主要用来获取网络内容，而 beautifulsoup4 库主要用来分析网络内容。 （ ）

10. 在 sqlite3 中，所有 SQL 语句的执行都要在游标对象的参与下完成。 （ ）

三、程序设计

1. 利用 tkinter 模块设计一个具有简单功能的计算器。

2. 设计一个文本编辑器界面，要求有打开、保存、新建等基本功能，并能进行基本的文字编辑。

参考文献

[1] 嵩天，礼欣，黄天羽. Python 语言程序设计基础[M]. 2 版. 北京：高等教育出版社，2017.

[2] Wesley Chun. Python 核心编程[M]. 3 版. 孙波翔，李斌，李晗，译. 北京：人民邮电出版社，2016.

[3] Zed A Shaw. "笨办法"学 Python[M]. 3 版. 王巍巍，译. 北京：人民邮电出版社，2014.

[4] 韦玮. 精通 Python 网络爬虫：核心技术、框架与项目实战[M]. 北京：机械工业出版社，2017.

[5] 董付国. Python 可以这样学[M]. 北京：清华大学出版社，2017.

[6] 张志强，赵越，等. 零基础学 Python[M]. 北京：机械工业出版社，2015.

[7] 廖雪峰. Python3 教程[EB/OL]. [2017-05].
http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000.

[8] 林信良. Python 程序设计教程[M]. 北京：清华大学出版社，2016.

[9] 李佳宇. 零基础入门学习 Python[M]. 北京：清华大学出版社，2016.

[10] 胡松涛. Python 网络爬虫实战[M]. 北京：清华大学出版社，2016.

[11] 董付国. Python 程序设计[M]. 2 版. 北京：清华大学出版社，2016.

[12] 邱仲潘，刘燕文，王水德. Python 程序设计教程[M]. 北京：清华大学出版社，2016.

[13] 周元哲. Python 程序设计基础[M]. 北京:清华大学出版社，2015.

[14] 王学颖，等. C++ 程序设计案例教程[M]. 2 版. 北京：科学出版社，2015.

[15] Python 官方网站[EB/OL]. [2017-05]. https://www.python.org.

[16] Vamei. Python 快速教程[EB/OL]. [2017-05]. https://www.shiyanlou.com/courses/214.

[17] 菜鸟教程. Python 基础教程[EB/OL]. [2017-05].
http://www.runoob.com/python/python-tutorial.html.

[18] 崔庆才. Python 爬虫学习系列教程[EB/OL]. [2017-05]. http://cuiqingcai.com/1052.html.